Lecture Notes for CSCI 3110: Design and Analysis of Algorithms

Travis Gagie
Faculty of Computer Science
Dalhousie University

Summer 2021

Contents

Chapter 1

“Clink” versus “BOOM”

Imagine I’m standing in front of you holding up two little pieces of silver-grey metal and between us is a bucket of water. You can see the pieces of metal look pretty much the same; I tap them on the desk so you can tell they sound about the same; I pass them around so you can check they feel about the same, weigh about the same, smell the same (not really at all), although I warn you not to taste them. I take them back and throw first one into the bucket, where it goes “clink”, and then the other, where it goes “BOOM”.

Even if we had in-person classes I probably wouldn’t be allowed to explode things in front of my students — that’s what tenure is for! — but I want you to think about how two things that seem the same can behave so differently. Explaining that for those two little pieces of metal gets us into discussions that get pretty deep pretty quickly, of chemical reactions and molecules and atoms. The famous physicist Richard Feynman said that if some disaster were going to wipe out nearly all scientific knowledge and he could choose once sentence to pass on to future generations, it would be “all things are made of atoms”. Other scientists might choose deep statements from their own fields that have changed humanity’s view of the universe and our place in it: for an astronomer, “the earth goes around the sun”; for a biologist, “species arise through a process of natural selection”; for a psychologist, “the mind is a function of the brain”. Statements about mathematics usually aren’t quite so controversial, at least among non-mathematicians, but the story goes that the first mathematician to prove that the square root of 2 is irrational was thrown overboard by his fellow Pythagoreans.

So, what have we got? You’ve been studying computer science in university for a few years now, and I’d like to know what you’ve learned about our field that can reasonably be considered one of the great scientific truths. “The speed of processors doubles every 18 months”? “Adding workers to a late software project makes it later”? “All your bases are belong to us”? “The cake is a lie”? This is when I’d really like to be able to stop and let you think, and then hear what you have to say. Under the circumstances, however, I’m just going to have to assume most of you chose to study computer science because it’s fun, you can do cool things with it (including saving people’s lives and making the world a better place), it pays pretty well and it’s indoor work with no heavy lifting — even though it may not tell us deep things about the universe.

That's rather a shame. If you check *Scientific American*'s list of "100 or so Books that Shaped a Century of Science", under "Physical Sciences" and mixed in with Stephen Hawking's *A Brief History of Time*, Primo Levi's *The Periodic Table*, Carl Sagan's *The Pale Blue Dot*, Paul Dirac's *Quantum Mechanics*, *The Collected Papers of Albert Einstein*, Linus Pauling's *Nature of the Chemical Bond* and Feynman's *QED*, you'll find books about computer science (or things close to it): Benoit Mandelbrot's *Fractals*, Norbert Weiner's *Cybernetics* and, last but definitely not least, Knuth's *Art of Computer Programming.* So apparently we are doing deep things, we're just not teaching you about them.

Actually, I think that in a very important sense, we are the deepest of the sciences. To understand where the basic rules of psychology come from, for example, you must know something about biology; to understand where the basic rules of biology come from, you must know something about chemistry; to understand where the basic rules of chemistry come from, you must know something about physics; to understand where the basic rules of physics come from, you must know something about mathematics; but who tells the mathematicians what they can and cannot do? *We do.* Philosophers, theologians and politicians might claim to, but I think the mathematicians largely ignore them. One field has really stood up to mathematics and laid down the law, and that field was computer science.

At the International Congress of Mathematicians in 1900, a famous mathematician named David Hilbert proposed ten problems that he thought mathematicians should work on in the 20th century. He later expanded the list to 23 problems, which were published in 1902 in the *Bulletin of the American Mathematical Society.* Let's talk about Hilbert's Tenth Problem (for the integers): "to devise a process according to which it can be determined in a finite number of operations whether [a given Diophantine equation with integer coefficients] is solvable in [the integers]". Notice Hilbert didn't ask whether such a process existed; back then, people assumed that if a mathematical statement were true, with enough effort they should be able to prove it.

I'll assume you all know what a quadratic equation in one variable is, and that you can tell if one has a solution in the reals using the quadratic formula. There are similar but more complicated formulas for solving cubic and quartic equations in one variable, which fortunately I never learned. If you're given a quintic equation in one variable with integer coefficients, then you may be able to use the rational-root theorem (which I actually did learn once upon a time!) to get it down to a quartic, which you can then solve with the quartic formulas. (For anything more complicated, maybe you can try Galois Theory, but I can't help you there.) Anyway, the Diophantine equations Hilbert was talking about are sort of like these equations but with a *finite* number of variables. He wanted a way to solve any given Diophantine equation that has integer coefficients or, at least, a way to tell if it even has an integer solution.

So, solving Diophantine equations over the integers was considered an important mathematics problem by important mathematicians. Unfortunately for them, in 1970 Yuri Matiyasevich showed in his doctoral thesis that we can take a certain general model of a computer, called a Turing Machine, and encode it as a Diophantine equation with integer coefficients, in such a way that the Turing Machine halts if and only if the Diophantine equation has an integer solution. It was already known that determining whether a Turing Machine halts is generally incomputable, meaning there is no algorithm for it (and that had already thrown a spanner in Hilbert's plan for 20th-century mathematics), so Matiyasevich's Theorem (also known as a Matiyasevich-Robinson-Davis-Putnam

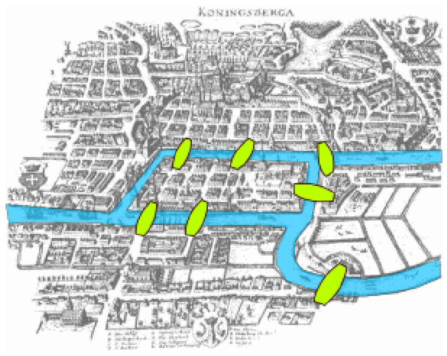


Figure 1.1: The seven bridges of Königsberg in Euler's time (`https://commons.wikimedia.org/wiki/File:Konigsberg_bridges.png`).

Theorem, for some other mathematicians who laid the groundwork) says Hilbert's Tenth Problem is impossible (for the integers). So, computer science told mathematics that it couldn't have something it really wanted.

I guess 1970 seems like a long time ago and we don't want to relive old victories so — although we'll talk a bit about Turing Machines and the Halting Problem and computability and Hilbert's *Entscheidungsproblem* — in this course mostly we're *not* going to focus on the gap between *possible* and *impossible* problems (and I won't mention Diophantine equations again). Instead, we're going to focus on the nature of the gap between *easy* problems (we have a polynomial-time solution that runs on a standard computer) and *hard* problems (we don't) — which is the deepest open question in computer science and has been for a long time, since 1971. Echoing Hilbert, in 2000 the Clay Mathematics Institute proposed a list of seven problems they thought mathematicians should work on in the 21st century, including this one of easy versus hard, and backed each with a million-dollar prize. To quickly illustrate the gap between easy and hard, I'd like to talk about two classic problems, determining whether a graph has an Eulerian cycle and whether it has a Hamiltonian cycle. At first these problems seem almost the same, but it turns out that one goes "clink" and the other goes "BOOM" (we think).

In the 1730s, the citizens of Königsberg (then in Prussia, now the Russian city of Kaliningrad) wrote to the famous mathematician Leonhard Euler asking whether it was possible to start somewhere in their city (shown as it existed then in Figure 1.1), walk around it in such a way as to cross each of their seven bridges once and only once, and arrive back at the starting point. Euler answered that it was not possible since, if we draw a graph with each shore and island as a vertex and each bridge as an edge, all of the vertices have odd degree, where a vertex's degree is the number of edges incident to it. He observed that in such a walk — now called an *Eulerian cycle* of a graph — every time we visit a vertex we reduce the number of uncrossed edges incident to it by 2 (we arrive across one and leave across one); therefore, in order for a graph to have an Eulerian cycle, it is a *necessary* condition that it be connected (that is, it must be possible to get from any vertex to any other vertex) and every vertex must have even degree.

Euler also showed that this is also a *sufficient* condition, which is less obvious. Suppose we start at some vertex v in a graph G that satisfies the condition, and walk around until we get stuck (that is, we arrive at a vertex all of whose incident edges we've already crossed). Where can we get stuck? Well, when we're at a vertex $w \neq v$ then we'll have "crossed off" an odd number of edges incident to w (one for each time we've arrived there and one for each time we've left, and we've arrived once more often than we've left); therefore, since an even number minus an odd number is an odd number and 0 isn't odd, there's always at least one edge left for us to leave across. It follows that we can only get stuck back at v, after tracing out a cycle C in G.

Let G' be the graph resulting from deleting from G the edges in C. If v has positive degree in G' then we can start there again and find and delete another cycle; otherwise, if some other vertex has positive degree in G', we can start there and find and delete another cycle. If we keep doing this until there are no edges left, we get a decomposition of G into edge-disjoint cycles (meaning the cycles can share vertices but not edges). Take any of the cycles C_1, choose a vertex v it shares with any of the other cycles, and choose another cycle C_2 containing v. (If C_1 doesn't share any vertices with any other cycles, then its vertices aren't connected to the rest of the graph, contrary to our assumption.) Replace C_1 and C_2 by a single (non-simple) cycle C_3 that makes a figure 8 in some sense: if we start at v then we first cross all the edges in C_1 and return to v, then cross all the edges in C_2 and return to v. Repeating this cycle-merging, we eventually obtain an Eulerian cycle of the graph.

It's not totally trivial to implement this algorithm cleanly but it's not all that hard either (so I gave it as a homework exercise last year) and it obviously takes only polynomial time in the size of the graph. In other words, the problem of determining whether a graph is Eulerian (contains an Eulerian cycle) goes "clink". It's actually an important problem that comes up in *de novo* genome assembly: one of the most common ways to assemble DNA reads is by building what's called a de Bruijn graph of the k-mers they contain, for some appropriate value of k, and then trying to find an Eulerian tour of that graph ("tour" instead of "cycle" because it need not end where it started). It's likely, for example, that this played a role in sequencing the Covid genome, which was important for developing a test for whether people were infected. Now let's consider a problem which seems deceptively similar at first.

In 1857 William Rowan Hamilton showed how to walk along the edges of a dodecahedron, visiting each vertex once and only once, until arriving back at the starting point. Such a walk in a graph is now called a *Hamiltonian cycle*, and it's an obvious counterpart to an Eulerian cycle but focusing on visiting vertices instead crossing edges. Hamilton didn't give a general result like Euler's, however, and in fact no one has ever found an efficient algorithm for determining whether a given graph is Hamiltonian (that is, it has a Hamiltonian cycle). By "efficient", in computer science we mean that the algorithm takes time polynomial in the size of the graph (in this case, the sum of the number of vertices and the number of edges). Obviously there are necessary conditions we can check quickly (for example, the graph has to be connected), and there are sufficient conditions we can check quickly (for example, if a graph is a clique then it's Hamiltonian, meaning it contains a Hamiltonian cycle), and there are some kinds of graphs that are easy (for example, cycles or trees), and some graphs are small enough that we can deal with them by brute force... but nobody's ever found an algorithm that can handle all graphs in polynomial time. In other words, it seems this problem goes "BOOM".

To see why people would really like to be able to determine whether a graph is Hamiltonian, suppose you're in charge of logistics for a delivery company and you're looking at a map trying to plan the route of a delivery truck that has to visit n cities, in any order and then return home. You can tell the distance between each pair of cities and you want to know if it's possible for the truck to make its deliveries while travelling at most a certain total distance. (This problem is called the TRAVELLING SALESPERSON PROBLEM or TSP or, assuming the truck travels only in the plane, EUCLIDIAN TSP; computer scientists often use SMALL CAPS for tricky problems.) If every road between two cities has distance 1, then the truck can travel distance n if and only if the graph with cities as vertices and roads as edges, is Hamiltonian. It follows that if HAMCYCLE (the problem of determining whether a graph is Hamiltonian) goes "BOOM", then TSP goes "BOOM" too. That is, if HAMCYCLE is hard, then so is TSP, because in some sense HAMCYCLE can be turned into TSP. Don't worry, we'll see *a lot* more about this later in the course.

How can two problems — determining if a graph is Eulerian and if it's Hamiltonian — that seem so similar be so different? Well, we hope investigating that will lead us to a deep truth, like investigating the reactions of those pieces of metal to water could lead us to studying atoms. We may not even really care as much about the problems themselves as we do about what studying them can tell us: my chemistry teacher *might* forgive me if I forgot that sodium metal explodes in water, but he'd be really annoyed if I forgot the atomic theory; similarly, you *may* pass this course even if you forget Euler's algorithm, but I really do want you to remember something about the difference between easy and hard problems.

Another way of saying this is that you shouldn't miss the forest for the trees. Think of Euler's Algorithm as a tree and HAMCYCLE as a tree and yourself as a student of geography: you're not really interested in individual trees as much as you're interested in the general layout of the forest and, metaphorically, whether there's a big river splitting it into two pieces. In the next class, we'll talk about another classic problem, colouring graphs, some versions of which (1-colourability, 2-colourability, planar k-colourability for $k \geq 4$) are easy and some versions of which (planar 3-colourability, general k-colourability for $k \geq 3$) are thought to be hard. Does anyone really care about 47-colourability in particular, say? Probably not, but by the end of this course you should understand that it's on the same side of the river as HAMCYCLE and TSP and planar 3-colourability and general k-colourability for any other $k \geq 3$ (that is, the "BOOM" side, to mix metaphors), and on the opposite side from finding Eulerian cycles and 1- or 2-colourability (the "clink" side).

Investigating the difference between the complexity of finding Eulerian and Hamiltonian cycles and between 2-colouring and 3-colouring graphs also leads us to a shallow but important truth about this course: it's dangerous to answer questions just by pattern matching! That is, just because a problem sounds similar to something you've seen before, don't think you can just write the answer to the old problem and get part marks.

Summing up, here are some things to remember while taking this course:

- Computer science is the deepest of the sciences!
- Some problems are possible, some aren't — and that's *deep*.
- Some problems are easy, some probably aren't — and that's *deep*, too.
- Don't miss the forest for the trees.
- It's dangerous to answer questions just by pattern matching.

Part I

Divide and Conquer

Chapter 2

Colouring Graphs

When I was in primary school geography was mainly about colouring maps, which I was pretty good at except that I kept loosing my pencil crayons, so my maps weren't as colourful as the other children's. I was therefore really impressed when I learned that in 1852 Francis Guthrie found a way to colour a map of the counties of England (there were 39 back then) using only four colours, such that no two counties with the same colour share a border (although meeting at a point is ok). This made him wonder if it was possible to 4-colour any (planar) map like this. He wrote to his brother, who was studying with the mathematician De Morgan (who formulated De Morgan's Laws of propositional logic, which you may have heard about last year), and the conjecture was published in *The Athenæum* magazine in 1854 and 1860. In 1890 Heawood proved any planar map can be 5-coloured, and in 1976 Appel and Haken finally proved the original conjecture by checking 1834 cases with a computer.

(An important (and very long) part of the proof was checked manually by Haken's daughter Dorothea, who taught me third-year algorithms at Queen's 22 years later; she's also a coauthor of the Master Theorem that we'll discuss next week.)

Appel and Haken's Four-Colour Theorem is considered part of graph theory, instead of geography, because if someone gives us a planar map and we draw a vertex in each region and put an edge between two vertices if and only if their regions share a border, then we get a planar graph that can be coloured with k colours such that no two vertices with the same colour share an edge, if and only if the original map can be coloured with k colours. For example, Figure 2.1 shows the graph we get for the map of the counties of Nova Scotia. (Actually, if we put an edge for every shared border — so one edge between France and Spain for their border west of Andorra and another for their border east of it, for example — then the graph and the map are *duals*, meaning that from the graph we can reconstruct something that's topologically equivalent to the map, and from that we can reconstruct the graph, and so on.) It's easy to find planar graphs that require 4 colours, such as a 4-clique (a graph on four vertices with edges between every pair), but it's not so easy to tell whether a given planar graph can be 3-coloured. In fact, as mentioned in the last lecture, that's a "BOOM" problem.

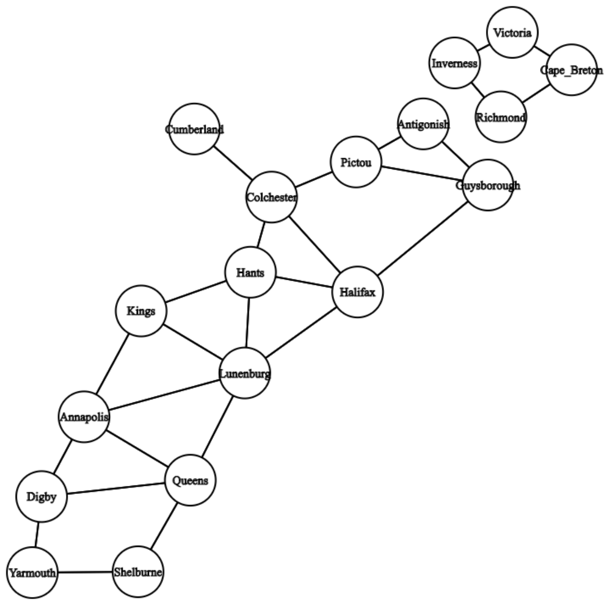


Figure 2.1: A graph that is 3-colourable if and only if the map of the counties of Nova Scotia is 3-colourable.

Suppose someone gives us a planar graph, meaning one that can be drawn in the plane such that edges don't cross, and asks us if it can be coloured with k colours. (There are polynomial-time algorithms for determining whether a graph is planar and, if so, finding a way to draw it without edge-crossings, so we won't worry about that.) If $k = 1$ then the answer is "yes" if and only if the graph contains no edges. If $k = 2$ then there's a simple algorithm based on breadth-first search: choose any vertex and colour it blue; colour its neighbours red; colour their uncoloured neighbours blue; keep going until the whole graph is coloured; if there's an edge with endpoints the same colour, tne answer is "no" and otherwise it's "yes". If $k \geq 4$ then, by the Four-Colour Theorem and the fact the graph is planar, the answer is "yes". No one has ever found an algorithm, however, for determining whether a graph is 3-colourable, whether or not it's restricted to be planar, that runs in polynomial time in the size of the graph in the worst case.

Of course, just because we don't have an algorithm that runs quickly in the worst case, doesn't mean we have to give up. For small graphs, we can just use brute force. Even for large graphs, it may be that almost all graphs are easy. We're going to explore that in this class, along with the idea of "degrees of BOOM", and then finally we'll end with some programming (which will probably be a relief by then).

Notice we didn't use the fact that the graph was planar in our tests for 1- and 2-colourability, so those problems are easy for general graphs too. If we can solve 3-Colourability (affectionately known as 3-Col) for general graphs, then obviously we can solve it for planar graphs as well. Later in the course we'll see that if we can solve Planar 3-Col in polynomial time then we can use it to solve general 3-Col. Therefore, Planar 3-Col goes "BOOM" if and only if general 3-Col goes "BOOM" — even though the latter is probably a bigger "BOOM". (Don't worry, by the end of the course you'll know the technical names for "clink" problems and "BOOM" problems.) When we consider 4-colourability, though, there's a big difference: while any planar graph can be 4-coloured, not every graph can be (think of a 5-clique).

In fact, if we could solve general 4-Col in polynomial time, then we could also solve 3-Col (and, thus, Planar 3-Col) in polynomial time. To see why, suppose we've been asked if a graph G is 3-colourable and we have a magic box that tells us quickly whether a graph is 4-colourable. If we build a graph G' by adding a vertex v to G and putting edges between v and all the original vertices of G, then G' is 4-colourable if and only if G is 3-colourable: if G' is 4-colourable and we take a 4-colouring of it and remove v, then the remainder — which is G — must be 3-coloured, because v couldn't have been the same colour as any other vertex; if G is 3-colourable and we take a 3-colouring of it and add v and colour v a new colour, then we get a 4-colouring of G'. In general, a polynomial-time algorithm for *general* $(k + 1)$-colourability immediately implies a polynomial-time algorithm for k-colourability.

Later in the course we'll see that if we have a polynomial-time algorithm for 3-Col (either planar or general, since I'll show you how to use the former to solve the latter) then we can use it to get a polynomial-time algorithm for a problem called Sat (determining whether a given propositional formula has a satisfying truth assignment). We'll then see that if we have such an algorithm for Sat then we can use it to get one for k-colourability for any k. We'll also see that if we have such an algorithm for Sat then we can use it to get one for HamCycle, and vice versa, and the same for TSP. This means that Sat goes "BOOM" if and only if 47-Col goes "BOOM", and that happens if and only if TSP goes "BOOM". If one of these problems is hard, they all are and, conversely, if we can solve one then we'll have solved them all.

This is probably all getting a little confusing by now, so I've drawn Figure 2.2 to help you visualize things, with an arrow from a problem u to a problem v if having a solution for v gives us a solution for u. The arrow goes from u to v instead of the other way around because, instead of changing the algorithm for v into an algorithm for u directly, what we normally do is design a polynomial-time algorithm that takes an instance of u and produces an instance of v such that the former is a "yes"-instance of u if and only if the latter is a "yes"-instance of v.

For example, the arrow from 3-Col to 4-col is because we saw how to take a graph G and produce a graph G' such that G is 3-colourable if and only if G' is 4-colourable. This is called a *reduction*. Note that the reduction can (and usually does) change the type: for example, when we reduce Sat to 3-Col later in the course, our reduction will turn a propositional formula into a graph. The solid arrows I should be able to test you on right now, and the dashed arrows I'll be able to test you on later in the course. We won't have time to see many reductions in this course, unfortunately, but there are *hundreds* of famous problems that all turn out to be essentially the same problem in different guises, and we don't know how to solve any of them quickly in the worst case.

Do you get the feeling we might be onto something deep here?

Now that we've talked about how we probably can't solve lots of interesting problems in polynomial time in the worst case, let's talk about how we can solve some of them in practice by *divide and conquer*. (Most people learn the divide-and-conquer approach to algorithm design for sorting

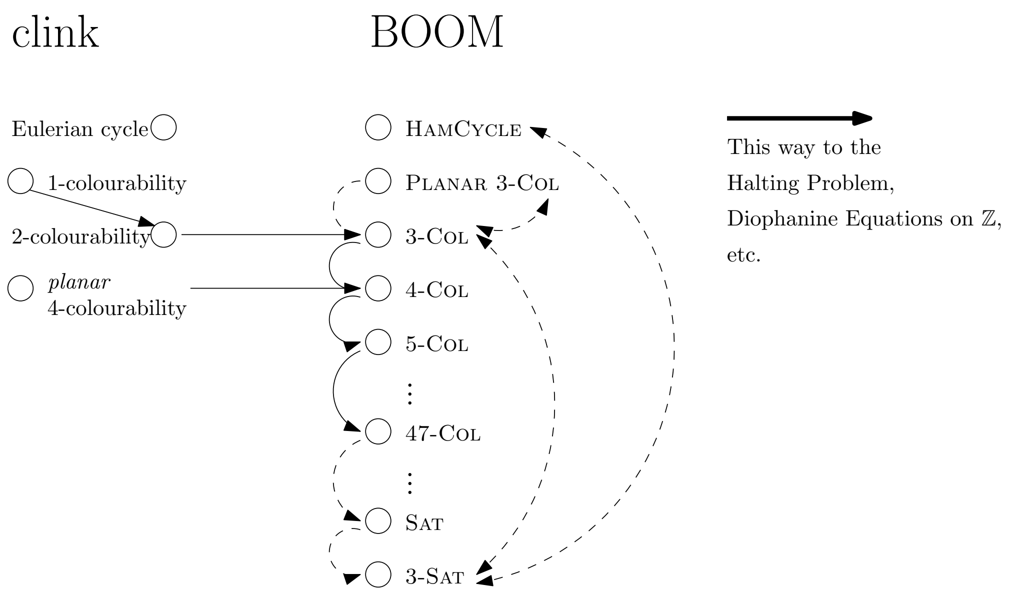


Figure 2.2: A *very* sketchy diagram of how some problems are related.

and thus get the impression it's only good for easy problems. I'm trying to avoid that with this course.) To start with, let's talk about Planar 3-Col or, rather, an even harder version of it: how many 3-colourings there are of the map of the 18 historical counties of Nova Scotia using only the colours red, green and yellow — such as the colouring shown in Figure 2.3? I've just told you that determining if there's at least one valid 3-colouring of a planar graph is a "BOOM" problem, but now I want to know exactly how many there are. Don't worry, it's not that bad: there are only 3^{18} possibilities to check, after all, and that's not even 400 million.

If we simply list the counties in alphabetical order, and for each list their neighbours, we get what's called the adjacency-list representation of the graph, as shown in Table 2.1. It's pretty easy to turn that into a simple program for counting 3-colourings, as shown in Figure 2.4, which might take a couple of seconds on a modern laptop. It hardly seems worth optimizing that, but it will be worth it when we get to bigger instances later.

The first optimization you might notice is that we're unnecessarily comparing Digby and Yarmounth twice (once as `Digby != Yarmouth` and again as `Yarmouth != Digby`). A more important optimization is that, since Cape Breton Island is an island, we can count its 3-colourings and the 3-colourings of the mainland separately, and multiply them to get the number of 3-colourings of the whole province, as shown in Figure 2.5. This way, instead of 3^{18} possibilities, we're only checking $3^4 + 3^{14}$, so our new program should be about 80 times faster.

It's not as obvious what to do next, since we don't have another big island to work with, but we can break the mainland into two "islands" of size 6 each if we consider Hants and Lunenberg first, as shown in Figure 2.6. The idea is that, for each way to colour Hants and Lunenberg, we count the ways to colour the mainland counties north of them, and the ways to colour the mainland counties south of them, and multiply those counts to get the number of ways of colouring the mainland

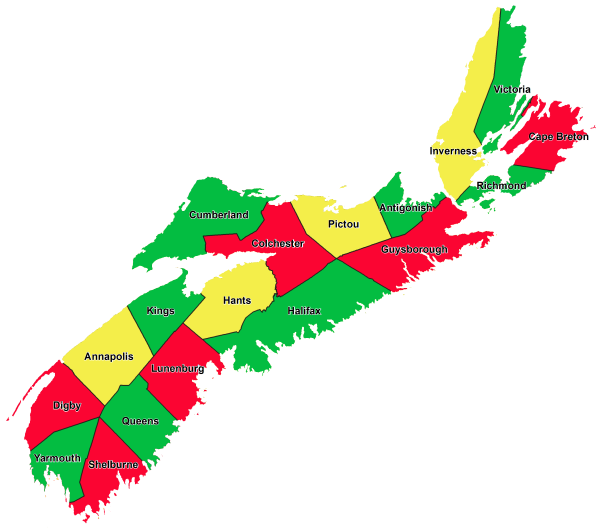


Figure 2.3: A 3-colouring of the counties of Nova Scotia.

given how we chose to colour Hants and Lunenberg. Summing over the 9 ways to colour Hants and Lunenberg (only 6 of which are valid!) we get the number of ways to colour the mainland. This way, we consider only $3^4 + 3^2 \cdot 2 \cdot 3^6 = 13203$ possibilities.

A final optimization is to check the endpoints of each edge are different colours as soon as we've coloured both endpoints. For example, in Figure 2.6, right after the line
`for (int Lunenberg = RED; Lunenberg <= YELLOW; Lunenberg++)`
we should check that Lunenberg is not the same colour as Hants,
`if (Lunenberg != Hants) { ... }` .

Just as Hants and Lunenberg break the mainland into two fairly small pieces, there's a theorem that says that in a planar graph on n vertices, we can always find a set of $O(\sqrt{n})$ vertices such that, when we remove them, none of the remaining connected pieces of the graph contain more than $2n/3$ vertices. The vertices we remove are called a *separator* and there are efficient algorithms for choosing them, but we're not going to see those in this course as, for graphs that aren't very big, it's usually possible to do it fairly well by hand. In any case, this result about separators gives us a recursive algorithm for counting the 3-colourings of planar graphs: find a separator and, for each way to colour the separator, recursively count the ways to colour each of the remaining pieces.

Writing a recurrence for this, we get something like

$$\begin{aligned} T(n) &= 3^{2\sqrt{n}} \cdot 2T(2n/3) \\ &= 3^{2\sqrt{n}} \cdot 2 \cdot 3^{2\sqrt{2n/3}} \cdot 2 \cdot 3^{2\sqrt{4n/9}} \cdots \\ &= 3^{O(n^{1/2} \log n)}, \end{aligned}$$

which isn't polynomial but is much better than $2^n n^{O(1)}$, which is what we get with the fastest known algorithm for finding the chromatic number of a general graph (the number of colours needed to colour it). Notice that it's not generally possible to choose a small separator in a general graph: no

Annapolis	Digby, Kings, Lunenburg, Queens
Antigonish	Guysborough, Pictou
Cape Breton	Richmond, Victoria
Colchester	Cumberland, Halifax, Hants, Pictou
Cumberland	Colchester
Digby	Annapolis, Queens, Yarmouth
Guysborough	Antigonish, Halifax, Pictou
Halifax	Colchester, Guysborough, Hants, Lunenburg
Hants	Colchester, Halifax, Kings, Lunenburg
Inverness	Richmond, Victoria
Kings	Annapolis, Hants, Lunenburg
Lunenburg	Annapolis, Halifax, Hants, Kings, Queens
Pictou	Antigonish, Colchester, Guysborough
Queens	Annapolis, Digby, Lunenburg, Shelburne
Richmond	Cape Breton, Inverness
Shelburne	Queens, Yarmouth
Victoria	Cape Breton, Inverness
Yarmouth	Digby, Shelburne

Table 2.1: The adjacency-list representation of the planar graph dual to the map of the counties of Nova Scotia.

matter how many vertices we remove from an n-clique, it remains connected (until we've removed the whole graph).

This all leads us to your homework, due in a little over a week: on Brightspace you'll find a text file with the adjacency-list representation of a graph corresponding to a simplified world map; figure out how many ways there are to 4-colour it. If you're feeling really enthusiastic, you can try to write general code that takes an encoding of a tree, such as the one shown in Figure 2.7, and uses it to compute the number of colourings: at a $\times$ node, start a new counter for each child, compute their values recursively, and multiply them; at a $+$ node, increment the current counter; at a node for a county, for each colour that isn't already assigned to one of the ancestors it's pointing at, assign it that colour and descend. (I may have made a mistake with the tree — it happens.) Honestly, any time you see that many nested for-loops, all doing essentially the same thing, you should think "There must be a better way!" (and maybe "Recursion!"). I'm not going to distribute a nice, general answer, because I want to use this exercise again (just switching maps).

Although the answer (and your code) are due in a week, I suggest you should try to get it done in the next 24 hours or so, while all of this is still fresh in your minds. Actually, I'm suggesting that only so I can finish the class by saying

Divide and Conquer! Today, Nova Scotia; tomorrow, the world!

```
#define RED 0
#define GREEN 1
#define YELLOW 2

int main() {
  int count = 0;

  for (int Annapolis = RED; Annapolis <= YELLOW; Annapolis++) [
    for (int Antigonish = RED; Antigonish <= YELLOW; Antigonish++) {
      ...
      for (int Yarmouth = RED; Yarmouth <= YELLOW; Yarmouth++) {
        if (Annapolis != Digby &&
          Annapolis != Kings &&
          Annapolis != Lunenberg &&
          Annapolis != Queens &&
          Antigonish != Guysborough &&
          Antigonish != Pictou &&
          ...
          Yarmouth != Digby &&
          Yarmouth != Shelburne) {
          count++;
        }
      }
      ...
    }
  }

  printf("There are %i ways to 3-colour Nova Scotia.\n", count);
  return(0);
}
```

Figure 2.4: A naïve program for counting the 3-colourings of Nova Scotia.

```
#define RED 0
#define GREEN 1
#define YELLOW 2

int main() {
  int CB_count = 0;
  int mainland_count = 0;

  for (Cape_Breton = RED; Cape_Breton <= YELLOW; Cape_Breton++) {
    for (Inverness = RED; Inverness <= YELLOW; Inverness++) {
      for (Richmond = RED; Richmond <= YELLOW; Richmond++) {
        for (Victoria = RED; Victoria <= YELLOW; Victoria++) {
          if (Cape_Breton != Victoria &&
            Cape_Breton != Richmod &&
            Inverness != Victoria &&
            Inverness != Richmond) {
            CB_count++;
          }
        }
      }
    }
  }

  for (int Annapolis = RED; Annapolis <= YELLOW; Annapolis++) [
    for (int Antigonish = RED; Antigonish <= YELLOW; Antigonish++) {
      ...
    }
  }

  printf("There are %i ways to 3-colour Nova Scotia.\n",
    CB_count * mainland_count);
  return(0);
}
```

Figure 2.5: A somewhat faster program for counting the 3-colourings of Nova Scotia. We have omitted the details of how to count the 3-colourings of the mainland, since they have not changed except that we leave out the counties on Cape Breton Island.

```
int mainland_count = 0;

for (int Hants = RED; Hants <= YELLOW; Hants++) {
  for (int Lunenberg = RED; Lunenberg <= YELLOW; Lunenberg++) {
    int north_count = 0;
    int south_count = 0;

    for (Antigonish = RED; Antigonish <= YELLOW; Antigonish++) {
      for (Colchester = RED; Colchester <= YELLOW; Colchester++) {
        ...
        for (Pictou = RED; Pictou <= YELLOW; Pictou++) {
          if (Hants != Colchester &&
            Hants != Halifax &&
            Hants != Lunenberg &&
            Lunenberg != Halifax &&
            Antigonish != Guysborough &&
            ...
            Guysborough != Pictou) {
            north_count++;
          }
        }
        ...
      }
    }

    for (Annapolis = RED; Annapolis <= YELLOW; Annapolis++) {
      for (Digby = RED; Digby <= YELLOW; Digby++) {
        ...
        for (Yarmouth = RED; Yarmouth <= YELLOW; Yarmouth++) {
          if (Hants != Kings &&
            Hants != Lunenberg &&
            Lunenberg != Annapolis &&
            Lunenberg != Kings &&
            Lunenberg != Queens &&
            Annapolis != Digby &&
            ...
            Shelburne != Yarmouth) {
            south_count++;
          }
        }
        ...
      }
    }

    mainland_count += north_count * south_count;
    }
  }
```

Figure 2.6: A somewhat faster way to compute the number of 3-colourings of the mainland.

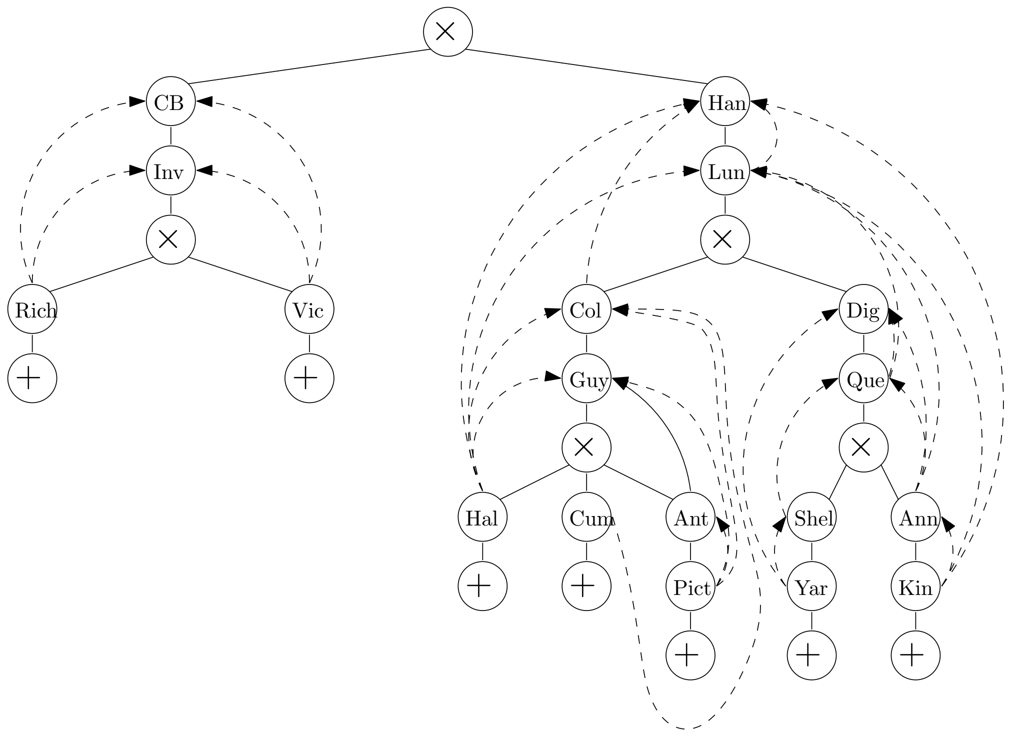


Figure 2.7: An abstract representation of a way to compute the number of 3-colourings of Nova Scotia.

Assignment 1

posted: 05.05.2021 **due: midnight 14.05.2021**

You can work in groups of up to three people. One group member should submit a copy of the solutions on Brightspace, with all members' names and banner numbers on it; the other group members should submit text files with all members' names and banner numbers (otherwise Brightspace won't let us assign them marks!). You may consult with other people but each group should understand the solutions: after discussions with people outside the groups, discard any notes and do something unrelated for an hour before writing up your solutions; it's a problem if no one in a group can explain one of their answers. For programming questions you should submit your code, which should compile and run correctly to receive full marks.

1. Use divide-and-conquer to count the valid 4-colourings of the map shown in Figure 2.8. (Apologies to New Zealand and Antarctica!) In this case, let's say a valid 4-colouring is an assignment of the colours red, green, yellow and purple to the regions of the map (ignore water and the current colours) such that no two regions assigned the same colour share a border of positive length (they can touch at a point) or have a line drawn between them. The adjacency-list representation of the corresponding graph (with the vertices names `A1` through `F4`) is on the back of this sheet and will be posted as a text file on Brightspace. If you find a discrepancy between the map and the adjacency lists, please email Travis as soon as possible!

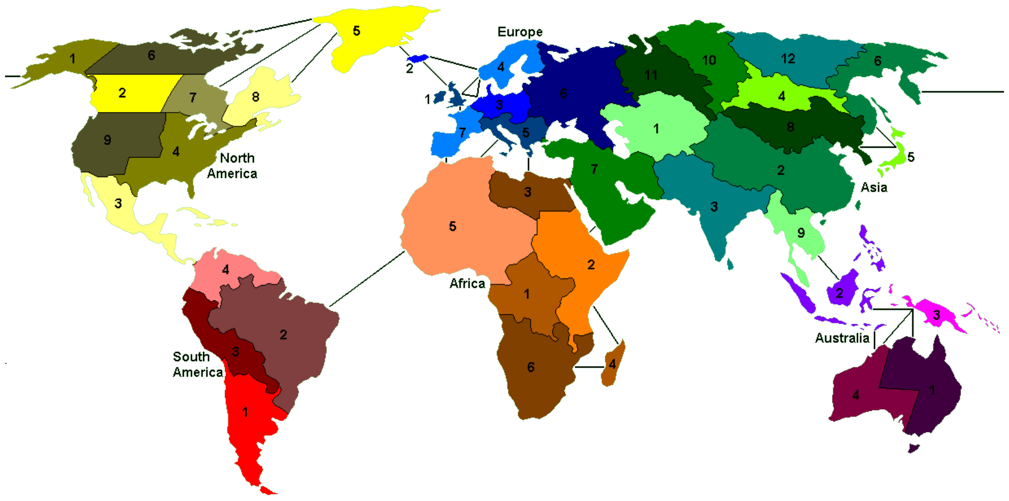


Figure 2.8: How many ways can this map (`https://commons.wikimedia.org/wiki/File:Risk_game_map_fixed.png`) be 4-coloured?

```
A1: A2, A5, A6
A2: A1, A3, A4, A5, A6, B7
A3: A2, A5, B7, D5
A4: A2, A6
A5: A1, A2, A3, D5, D7, F2
A6: A1, A2, A4

B1: B2, B3, B7, B11, D6
B2: B1, B3, B8, B9, B10, B11
B3: B1, B2, B7, B9
B4: B6, B8, B10, B12
B5: B6, B8
B6: B4, B5, B8, B12, E1
B7: A2, A3, B1, B3, D5, D6
B8: B2, B4, B5, B6, B10
B9: B2, B3, C2
B10: B2, B4, B8, B11, B12
B11: B1, B2, B10, D6
B12: B4, B6, B10

C1: C3, C4
C2: B9, C3, C4
C3: C1, C2, C4
C4: C1, C2, C3

D1: D2, D3, D4, D7
D2: D1, D4, E5
D3: D1, D4, D5, D6, D7
D4: D1, D2, D3, D6
D5: A3, A5, B7, D3, D6, D7
D6: B1, B7, B11, D3, D4, D5
D7: A5, D1, D3, D5

E1: B6, E2, E6
E2: E1, E6, E7, E9
E3: E4, E9, F4
E4: E3, E7, E8, E9
E5: D2, E6, E7, E8
E6: E1, E2, E5, E7
E7: E2, E4, E5, E6, E8, E9
E8: E4, E5, E7
E9: E2, E3, E4, E7

F1: F2, F3
F2: A5, F1, F3, F4
F3: F1, F2, F4
F4: E3, F2, F3
```

Chapter 3

Euclid, Karatsuba, Strassen

Euclid's algorithm is one of the oldest, having appeared in his *Elements* about 300 BCE, and says that given two integers a and b — for simplicity, assume $a > b > 0$ — we can find the greatest common divisor $\langle a, b\rangle$ of a and b by finding $c = a \bmod b$ and then recursively finding $\langle b, c\rangle$. For example,

$$\langle 168, 78\rangle = \langle 78, 12\rangle = \langle 12, 6\rangle = \langle 6, 0\rangle = 6.$$

Euclid actually proposed repeatedly subtracting the smaller of the two numbers from the larger one, but that can obviously be really slow: it takes $\lfloor a/b\rfloor$ step to do the same thing as a single mod step, and so the whole algorithm can take $\Omega(a)$ steps in the worst case, when a is huge and b is tiny. (We'll see Ω soon; for now, just think of O as meaning "roughly at most" and Ω as meaning "roughly at least".) Rather surprisingly, the complexity of the mod version wasn't analyzed until the 19th century, when it was shown to take $O(\log a)$ steps. To see why, consider than if $b < a/2$ then $a \bmod b < b < a/2$, and if $b \geq a/2$ then $a \bmod b \leq a - b < a/2$. Either way, at each step the sum of the two numbers we're considering decreases by at least half the larger number, which is at least a quarter of the sum, so the number of steps can't be more than about $\log_{4/3}(a + b)$. In fact, Lamé proved in 1844 that the number of steps is no more than 5 times the number of digits in the decimal representation of a, and this bound is tight when a and b are consecutive Fibonacci numbers. Lamé's proof is considered the beginning of computational complexity theory (the study of how long computations take).

I was careful to write "steps" in the preceding paragraph because it's not clear that taking the mod of two numbers takes constant time, especially not for a human. In just a moment we'll discuss how long basic arithmetic operations take, but first we should decide how we measure the size of the input (so we can express the complexity as a function of that size), what model of computation we're working in, and how to measure the difficulty of a problem.

When I wrote "$O(\log a)$ steps", I was expressing the complexity in terms of the numbers themselves (since $a > b$ means $O(\log a) = O(\log(a + b))$); Lamé's bound — linear in the number of digits in a — is in terms of the size of the input, with "size" interpreted as the amount of paper it would occupy when the numbers are written in base c for some constant $c > 1$. (Notice that choosing another constant base $d > 1$ would change the number of digits by only a constant factor, which disappears in the O, although the number of digits in unary — writing a as a copies of 1 — is exponentially larger.) Although there can be exceptions, that "size of the paper" measure is the default.

To remember this, think of factoring integers. You may have heard that factoring is believed to be hard, and we don't have a fast algorithm for it, at least not on classical computers. Probably the most famous quantum algorithm, Shor's algorithm, should be able to factor in polynomial time if we can build the hardware needed to run it.

(If you haven't heard that factoring is believed to be hard, I recommend the movie *Sneakers*, starring Robert Redford, Dan Aykroyd, Ben Kingsley, Mary McDonnell, River Phoenix, Sidney Poitier and David Strathairn), about what happens when a mathematician figures out how to factor quickly. Incidentally, I just read that Len Adleman — the A in the RSA public-key cryptosystem — prepared the lecture that's given in one of the scenes, in exchange for his wife meeting Robert Redford.)

How hard is it to factor a number n? Consider the following simple program and how many steps it takes (with `%` meaning mod in `C`):

```
for (int i = 2; i < n; i++) {
    if (n % i = 0) {
        printf("%i\n", i);
    }
}
```

Pretty clearly this takes $O(n)$ steps: linear in n, but exponential in the number of digits in n. This will come up later in the course when we discuss the difficulty of the "BOOM" problem KNAPSACK, which is NP-hard only in the weak sense, meaning it can be solved in time polynomial in the numbers involved but not (we think) in the number of digits in them. If a problem is NP-hard in the strong sense, then we don't know how to solve it in polynomial time even when the numbers are written in unary.

Usually, when we tell a computer to add two numbers, for example, we can assume it'll happen in a constant number of clock cycles. It's fairly rare to be dealing with numbers so big they won't fit in a machine word — normally 64 bits or at least 32 bits on any reasonably recent machine — and word-level parallelism allows us to do a lot with numbers that will fit. (Even the number of 4-colourings from your assignment fits in 64 bits!) Algorithm designers often work in the *word-RAM model* with words whose size is logarithmic in the size of the input. One reason for this is that, if an algorithm runs in time polynomial in the size of the input, then it also uses space (memory) polynomial in the size of the input, so pointers take a logarithmic number of bits and fit in a constant number of words. Operations on words are considered to take constant time if they can be computed by AC0 circuits, which have constant depth and a number of gates polynomial in the word-size (so polylogarithmic in the input size, in the word-RAM model). Addition and subtraction are in AC0 but multiplication and parity (checking whether the word contains an odd or even number of 1s) aren't, so some people still tweak their algorithms to avoid multiplications, even though multiplication is still fast in practice on modern machines. We'll usually assume we're working in the word-RAM model in this course, but we won't mention it much. If you go further in algorithmics, it'll become more important, but for now you just need to know that there is a formal definition of what's fast and what's not.

We're also usually going to consider the difficulty of a problem, as a function of the size of the input, to be the difficulty of the hardest instances as the size goes to infinity. That is, if we say we can compute a function of graphs on n vertices and m edges in $O(n+m)$ time, we mean that no matter how big n and m are, we can compute the function that quickly for any graph with n vertices and m edges. (We'll talk in a later class about exactly what O means and why it means we only have to worry about sufficiently large graphs.) This is *worst-case complexity*, and it's the default. You'll also hear about best-case, expected-case and average-case complexities. A lot of people seem to think O means "in the worst case" and Ω means "in the best case" but, as we'll discuss in that later class, they're confused.

So, how long does addition take for a human? Suppose I give you two n-digit numbers and you perform the standard algorithm for addition that we learn in primary school, working from right to left and carrying when necessary. It's easy to prove by induction that, with two numbers, the carry is never bigger than 1; therefore, you use $O(n)$ time. That's optimal, since you have at least to read the input.

Now, how long does multiplication take, for a human (and for a computer when the numbers are much bigger than the word size)? Again, suppose I give you two n-digit numbers and you perform the standard algorithm for multiplication that we learn in primary school, working from right to left in the second number and multiplying each digit by the whole first number, padding on the right with 0s, and then summing up all the products. This takes $O(n^2)$ time and $O(n^2)$ digits of space, using $O(n)$-time addition as a subroutine. (Actually, both those $O(n^2)$s should be $\Theta(n^2)$s, but we'll discuss that in another class.) Can we do better? For example,suppose we're multiplying 85419621 by 75339405:

```
        85419621
        75339405
        --------
       427098105
      0000000000
     34167848400
    768776589000
   2562588630000
  25625886300000
  427098105000000
 5979373470000000
 ----------------
 6435463421465505
```

First, it's not really hard to reduce the space to $O(n)$, by figuring out the digits in the addition column-wise instead of row-wise. For example, suppose we've worked out that last 8 digits in the answer are 21465505 and the carry into the ninth column from the right in the addition is 4, all of which takes $O(n)$ digits of space to store. Also, suppose we know that the carries are as follows: 4

for the multiplication by the 5 on the right (because $5 \times 8 + 2 = 42$); 0 for the multiplication by 0; 1 for the multiplication by 4; 1 for the multiplication by 9; 2 for the multiplication by the 3 on the right; 1 for the multiplication by the 3 on the left; 1 for the multiplication by the 5 on the left; and 0 for the multiplication by 7. These carries also take $O(n)$ digits of space to store.

There are no more digits in 85419621 to multiply the 5 on the right by, so its entry in the ninth column from the right is just its carry, 4; the entry for 0 is $(0 \times 8 + 0) \bmod 10 = 0$; the entry for 4 is $(4 \times 5 + 1) \bmod 10 = 1$ and its next carry is $\lfloor (4 \times 5 + 1)/10 \rfloor = 2$; the entry for 9 is $(9 \times 4 + 1) \bmod 10 = 7$ and its next carry is $\lfloor (9 \times 4 + 1)/10 \rfloor = 3$; the entry for the 3 on the right is $(3 \times 1 + 2) \bmod 10 = 5$ and its carry is $\lfloor (3 \times 1 + 2)/10 \rfloor = 0$; the entry for the 3 on the left is $(3 + 1) \bmod 10 = 8$ and its next carry is $\lfloor (3 + 1)/10 \rfloor = 2$; the entry for the 5 on the left is $(5 \times 6 + 1) \bmod 10 = 1$ and its next carry is $\lfloor (5 \times 6 + 1)/10 \rfloor = 3$; finally, the entry for 7 is $(7 \times 2 + 0) \bmod 10 = 4$ and its next carry is $\lfloor (7 \times 2 + 0)/10 \rfloor = 1$. Summing up the carry into the 9th column of the addition and the entries, we get $4 + (4 + 0 + 1 + 7 + 5 + 8 + 1 + 4) = 34$, so the ninth digit from the right in the answer is 4 and the carry into the tenth column from the right in the addition is 3. Now we can forget the entries in the ninth column and just remember the carries and the last 9 digits of the answer, 421465505. This still takes $O(n^2)$ time but at least we never need to store more than $O(n)$ digits of space.

There are lots of folklore methods for fast multiplication, but I don't think any of them actually beat that $O(n^2)$ time bound. At least, in a seminar in the 1960, the famous mathematician Andrei Kolmogorov (who'll come up again when we get to Kolmogorov complexity and why it's incomputable) conjectured that $\Omega(n^2)$ is a lower bound for multiplication. A student in the seminar, Anatoly Karatsuba, thought about the problem for a few days and then presented Kolmogorov with a simple divide-and-conquer algorithm that takes subquadratic time, disproving Kolmogorov's conjecture. Kolmogorov was so impressed that he presented the algorithm in lectures, wrote it up and sent it to a journal with Karasuba's name on it.

Suppose we are given two n-digit numbers x and y to multiply and, for simplicity, assume x and y are written in decimal and n is a power of 2. If n is 1, we can work out xy in constant time, so suppose $n > 1$. We rewrite xy as

$$(x_1 \cdot 10^{n/2} + x_0)(y_1 \cdot 10^{n/2} + y_0)\,,$$

where x_1 and y_1 consist of the first $n/2$ digits of x and y respectively, and x_0 and y_0 consist of the last $n/2$ digits of s and y respectively. The obvious way to work this out,

$$(x_1 \cdot 10^{n/2} + x_0)(y_1 \cdot 10^{n/2} + y_0) = x_1 y_1 \cdot 10^n + (x_1 y_0 + x_0 y_1) \cdot 10^{n/2} + x_0 y_0$$

seems to require four multiplications of $(n/2)$-digit numbers, plus some easy shifts and additions, which doesn't help since $4(n/2)^2 = n^2$. However, Karatsuba noticed that if we define

$$\begin{aligned} z_2 &= x_1 y_1\,, \\ z_1 &= x_1 y_0 + x_0 y_1\,, \\ z_0 &= x_0 y_0\,, \end{aligned}$$

so

$$xy = z_2 \cdot 10^n + z_1 \cdot 10^{n/2} + z_0\,,$$

then since

$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0\,,$$

we can use one multiplication to work out z_2, one multiplication to work out z_0, and then one multiplication (and two additions and two subtractions, which are easy) to work out z_1.

(You might wonder if $x_1 + x_0$ and $y_1 + y_0$ couldn't be $(n/2 + 1)$-digit numbers. They could, but let's ignore that and get through this lecture. Karatsuba's algorithm doesn't actually need the number of digits in the numbers to be powers of 2, or equal, since we can always pad on the left with 0s.)

Of course just saving a quarter of the multiplications doesn't improve the asymptotic bound; for that, we have to apply this technique recursively, working out each multiplication of two $(n/2)$-digit numbers using three multiplications of $(n/4)$-digit numbers. This gives us a recurrence

$$T(n) = 3T(n/2) + 2n\,,$$

where the $2n$ is just my guess at the the complexity of all the additions and subtractions, and doesn't really matter. Expanding this, we get

$$\begin{aligned} T(n) &= 3T(n/2) + 2n \\ &= 3(3T(n/4) + n/2) + 2n \\ &= 3(3(3T(n/8) + 2(n/4)) + 2(n/2)) + 2n \\ &\vdots \end{aligned}$$

so expanding i times gives us

$$T(n) = 3^i T(n/2^i) + 2n \sum_{j=0}^{i-1} (3/2)^j\,.$$

If we set $i = \lg n$ and assume $T(1) = 1$, where lg denotes $\log_2$, then since

$$2n \sum_{j=0}^{\lg n - 1} (3/2)^j < 2n(3/2)^{\lg n - 1} \sum_{k \geq 0} (2/3)^k = O(3^{\lg n})\,,$$

we get $T(n) = O(3^{\lg n}) \subset O(n^{1.585})$.

For example, to multiply 85419621 and 75339405 with Karatsuba's algorithm, we'd compute

$$\begin{aligned} z_2 &= 8541 \times 7533 = 64339353\,, \\ z_0 &= 9621 \times 9405 = 90485505\,, \\ z_1 &= (8541 + 9621)(7533 + 9405) - z_2 - z_0 \\ &= 18162 \times 16938 - 64339353 - 90485505 \\ &= 307627956 - 154824858 \\ &= 152803098 \end{aligned}$$

and then

$$6433935300000000 + 1528030980000 + 90485505 = 6435463421465505\,.$$

As you can probably guess, it doesn't really make sense to use Karatsuba's algorithm for any except really huge numbers. (It's also worth noting that, although Karatsuba's algorithm is recursive, just because you start breaking up big numbers and multiplying the pieces doesn't mean you're stuck doing that; once the pieces get small enough, you can switch to the standard algorithm.) Still, I like teaching Karatsuba's algorithm for several reasons:

- Who would have guess that after thousands of years of people multiplying in quadratic time, multiplication would finally be sped up in 1962?
- Stephen Cook (one of my professors at Toronto, who'll also come up again in this course) studied the complexity of multiplication for his PhD thesis, and a generalization of Karatsuba's algorithm (splitting each number into k parts) is named Toom-Cook.
- In March 2019 — just over two years ago, and just six months before the first time I taught this course! — Harvey and van der Hoeven gave an $O(n \log n)$-time algorithm for multiplying two n-digit numbers.
- Karatsuba's algorithm is a gentle introduction to the ideas used in Strassen's algorithm for matrix multiplication — which is kind of mysterious otherwise — and the matrix-multiplication exponent is sort of theoretical computer science's answer to the gravitational constant.

Once we know we can speed up multiplication of integers by divide-and-conquer, it's natural to wonder what other kinds of multiplication we can speed up. Strassen considered matrix multiplication, such as

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}.$$

In high school, we learned to get the value in cell (i, j) of C by multiplying the ith row of A by the jth column of B. If A, B and C are $n \times n$ matrices, however — with each of $A_{1,1}$ through $C_{2,2}$ being an $(n/2) \times (n/2)$ matrix — then this involves computing the dot product of two n-component matrices for each of the n^2 entries of C, and thus takes $\Omega(n^3)$ time overall, even assuming multiplying each pair of components takes constant time.

In 1969 Strassen took the equations we all know,

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1}\,, \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2}\,, \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1}\,, \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2}\,, \end{aligned}$$

and defined 7 intermediate matrices similar to Karatsuba's z_2, z_1, z_0 (but harder to memorize — you don't have to!):

$$\begin{aligned} M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})\,, \\ M_2 &= (A_{2,1} + A_{2,2})B_{1,1}\,, \end{aligned}$$

$$\begin{array}{rcl} M_3 & = & A_{1,1}(B_{1,2} - B_{2,2})\,, \\ M_4 & = & A_{2,2}(B_{2,1} - B_{1,1})\,, \\ M_5 & = & (A_{1,1} + A_{1,2})B_{2,2}\,, \\ M_6 & = & (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})\,, \\ M_7 & = & (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})\,. \end{array}$$

Once we've computed M_1 through M_7 using 7 multiplications of $(n/2) \times (n/2)$ matrices and a constant number of matrix additions and subtractions (which take $O(n^2)$ time each, assuming we can add and subtract components in constant time), then we can compute $C_{1,1}$ through $C_{2,2}$ using only a constant number more additions and subtractions, as follows:

$$\begin{array}{rcl} C_{1,1} & = & M_1 + M_4 - M_5 + M_7\,, \\ C_{1,2} & = & M_3 + M_5\,, \\ C_{2,1} & = & M_2 + M_4\,, \\ C_{2,2} & = & M_1 - M_2 + M_3 + M_6\,. \end{array}$$

Checking only $C_{1,1}$, for brevity, we see that indeed

$$\begin{array}{rcl} C_{1,1} & = & M_1 + M_4 - M_5 + M_7 \\ & = & (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) + A_{2,2}(B_{2,1} - B_{1,1}) - \\ & & (A_{1,1} + A_{1,2})B_{2,2} + (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \\ & = & A_{1,1}B_{1,1} + A_{1,1}B_{2,2} + A_{2,2}B_{1,1} + A_{2,2}B_{2,2} + A_{2,2}B_{2,1} - A_{2,2}B_{1,1} - \\ & & A_{1,1}B_{2,2} - A_{1,2}B_{2,2} + A_{1,2}B_{2,1} + A_{1,2}B_{2,2} - A_{2,2}B_{2,1} - A_{2,2}B_{2,2} \\ & = & A_{1,1}B_{1,1} + A_{1,2}B_{2,1}\,. \end{array}$$

Computing C from M_1 through M_7 saves us one multiplication compared to computing it directly from A and B, which leads to the recurrence

$$T(n) = 7T(n/2) + cn^2$$

for some constant c (for the additions and subtractions). Expanding and plugging in $i = \lg n$ as before,

$$\begin{array}{rcl} T(n) & = & 7T(n/2) + cn^2 \\ & = & 7(7T(n/4) + c(n/2)^2) + cn^2 \\ & = & 7(7(7T(n/8) + c(n/4)^2) + c(n/2)^2) + cn^2 \\ & \vdots & \end{array}$$

we get

$$T(n) = 7^{\lg n} + cn^2 \sum_{j=0}^{\lg n - 1} (7/4)^j\,.$$

Since

$$cn^2 \sum_{j=0}^{\lg n - 1} (7/4)^j < cn^2 (7/4)^{\lg n - 1} \sum_{k \geq 0} (4/7)^k = O(7^{\lg n})$$

we get

$$T(n) = O(7^{\lg n}) \subset O(n^{2.8074})\,.$$

That is, Strassen's algorithm multiplies matrices in subcubic time. You can read more about it on Wikipedia (`https://en.wikipedia.org/wiki/Strassen_algorithm`), which is where I just looked up the definitions of M_1 through M_7. I've taught Karatsuba's algorithm often enough that I've pretty much memorized $z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$, but if I ever get to the point where I've memorized the definitions of M_1 through M_7 then I think it'll be time to quit.

People actually do use Strassen's algorithm in practice for large enough matrices, padding with rows and/or columns of 0s to deal with matrices that aren't square or whose height and/or width are odd, although there are asymptotically faster algorithms. Currently the asymptotically fastest algorithm, published by Alman and Vassilevska-Williams in January this year, runs in $O(n^{2.3728596})$ time. Since matrix multiplication is used as a subroutine in many other algorithms (we'll see some later in the course), according to some people the question of what that exponent actually is stands as one of the big open problems in theoretical computer science.

Chapter 4

Fast Fourier Transform

We've looked at multiplying integers and square matrices quickly using divide-and-conquer, and those sets are two classic examples of rings: addition and multiplication are defined, there are additive and multiplicative identity elements and additive inverses, and multiplication is associative and distributive but may not be commutative. Another classic example of a ring is the set of polynomials, and today we're going to look at a divide-and-conquer algorithm, the Fast Fourier Transform (FFT), for multiplying those quickly.[*]

Since polynomials are functions, the most obvious thing to do with them is evaluate them. Given a degree-n polynomial

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

and a value for x, we can evaluate $A(x)$ in $O(n)$ time as follows:

```
double eval(int n, double *A, double x) {
  double powx = 1.0;
  double value = 0.0;

  for (int j = 0; j < n; j++) {
    value += A[j] * powx;
    powx *= x;
  }
  return(value);
}
```

[*]Gauss knew about a form of the FFT in the early 1800s but its widespread use dates from Cooley and Tukey's rediscovery of it in the 1960s, and it's now one of the most widely used numerical algorithms. I'll try to give a basic introduction here but, if you want more details, you should watch Max's lecture (actually, you should already have watched that), read the chapter on the FFT in *Introduction to Algorithms* or watch Demaine's lecture about it (`https://www.youtube.com/watch?v=iTMn0Kt18tg`) and read his lecture notes (`https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/lecture-notes/MIT6_046JS15_lec03.pdf`) — which is how I reviewed the FFT before writing this.

(This can be written more mathematically as a formula called Horner's Rule, but perhaps a program is clearer for us.)

If we want to add $A(x)$ and another degree-n polynomial

$$B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1} ,$$

then we can just add corresponding coefficients, also in $O(n)$ time:

$$A(x) + B(x) = (a_0 + b_0) + (a_1 + b_1)x_1 + (a_2 + b_2)x^2 + \cdots + (a_{n-1} + b_{n-1})x^{n-1} .$$

Now suppose we want to multiply $A(x)$ and $B(x)$. The standard way we were taught in high school involves $\Theta(n^2)$ multiplications of coefficients,

$$C(x) = A(x)B(x) = \sum_{j=0}^{2n-2} x^j \sum_{k=0}^{j} a_k b_{j-k} .$$

As you'll see in the next assignment, it's possible to speed that up using a version of Karatsuba's algorithm.

For example, if

$$\begin{array}{rcl} A(x) & = & 5 + 3x - 2x^2 + 4x^3 , \\ B(x) & = & -2 - x + 3x^2 + x^3 \end{array}$$

then

$$A(x) + B(x) = (5 - 2) + (3 - 1)x + (-2 + 3)x^2 + (4 + 1)x^3 = 3 + 2x + x^2 + 5x^3$$

but

$$\begin{array}{rcl} C(x) & = & A(x)B(x) \\ & = & (5 + 3x - 2x^2 + 4x^3)(-2 - x + 3x^2 + x^3) \\ & = & (-10 - 5x + 15x^2 + 5x^3) + \\ & & (-6x - 3x^2 + 9x^3 + 3x^4) + \\ & & (4x^2 + 2x^3 - 6x^4 - 2x^5) + \\ & & (-8x^3 - 4x^4 + 12x^5 + 4x^6) \\ & = & -10 + (-5 - 6)x + (15 - 3 + 4)x^2 + (5 + 9 + 2 - 8)x^3 + \\ & & (3 - 6 - 4)x^4 + (-2 + 12)x^5 + 4x^6 \\ & = & -10 - 11x + 16x^2 + 8x^3 - 7x^4 + 10x^5 + 4x^6 . \end{array}$$

The first idea behind the FFT is something like evaluating $A(x)$ and $B(x)$ at n distinct values $x_0, \ldots, x_{n-1}$; calculating $C(x_j) = A(x_j)B(x_j)$ for each x_j; and then recovering the coefficients of $C(x)$ from the pairs $(x_0, C(x_0)), \ldots, (x_{n-1}, C(x_{n-1}))$. This isn't quite right — because $C(x)$ is a polynomial of degree $2n$ and thus can be uniquely specified by $2n$ points on its curve but not by only n points — but it is if we assume (without loss of generality) that n is a power of 2 and the last $n/2$ coefficients in $A(x)$ and $B(x)$ are all 0s.

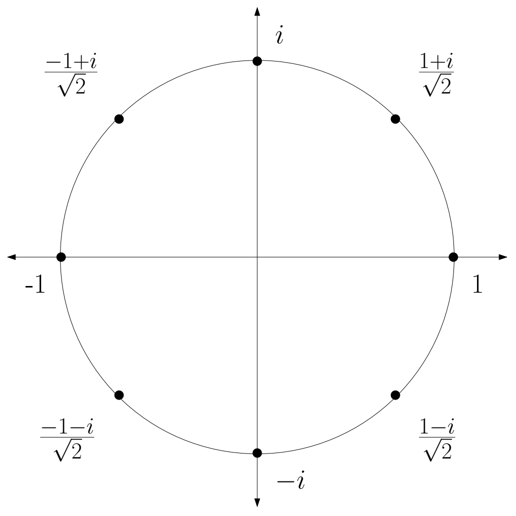


Figure 4.1: The 8th roots of unity.

The second idea is to choose $x_0, \ldots, x_{n-1}$ to be the n distinct complex numbers c such that $c^n = 1$ (called "the nth roots of unity"). Drawn in the complex plane, as in Figure 4.1 for $n = 8$, these are n points equally spaced around the unit circle, starting at $1 = 1 + 0i$ (which corresponds to $(1, 0)$ in the Euclidean plane).

To evaluate $A(x)$ at $x_0, \ldots, x_{n-1}$, we compute the matrix-vector product

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-2} & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-2} & x_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 1 & x_{n-2} & x_{n-2}^2 & \ldots & x_{n-2}^{n-2} & x_{n-2}^{n-1} \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-2} & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-2} \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} A(x_0) \\ A(x_1) \\ A(x_2) \\ \vdots \\ A(x_{n-2}) \\ A(x_{n-1}) \end{pmatrix}.$$

The matrix V on the left is a *Vandermonde matrix* (meaning the entries in each row are increasing powers) and we'll discuss it in detail next; the components of the vector $\vec{A}$ by which its multiplied are just the coefficients of $A(x)$; and the components of the product are $A(x)$ evaluated at $x_0, \ldots, x_{n-1}$.

To see why we choose $x_0, \ldots, x_{n-1}$ to be the nth roots of unity, consider V for the 8th roots of unity and $\vec{A}$ for our example polynomial $A(x)$:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & (1+i)/\sqrt{2} & i & (-1+i)/\sqrt{2} & -1 & (-1-i)/\sqrt{2} & -i & (1-i)/\sqrt{2} \\ 1 & (-1-i)/\sqrt{2} & i & (1-i)/\sqrt{2} & -1 & (1+i)/\sqrt{2} & -i & (-1+i)/\sqrt{2} \\ 1 & (-1+i)/\sqrt{2} & -i & (1+i)/\sqrt{2} & -1 & (1-i)/\sqrt{2} & i & (-1-i)/\sqrt{2} \\ 1 & (1-i)/\sqrt{2} & -i & (-1-i)/\sqrt{2} & -1 & (-1+i)/\sqrt{2} & i & (1+i)/\sqrt{2} \end{pmatrix} \begin{pmatrix} 5 \\ 3 \\ -2 \\ 4 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

There seems to be a lot of structure to V, doesn't there? Among other things, in our example it's an 8×8 matrix containing 8 distinct values. We'll use that structure to multiply V by $\vec{A}$ in $O(n \log n)$ time rather than $\Omega(n^2)$ time, which the high-school algorithm would use.

The third idea behind the FFT is that $A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2)$, where

$$\begin{aligned} A_{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{(n-2)/2} \\ A_{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{(n-2)/2} \end{aligned}$$

so

$$\begin{aligned} A_{\text{even}}(x^2) &= a_0 + a_2 x^2 + a_4 x^4 + \cdots + a_{n-2} x^{n-2} \\ x A_{\text{odd}}(x^2) &= a_1 x + a_3 x^3 + a_5 x^5 + \cdots + a_{n-1} x^{n-1}\,. \end{aligned}$$

We can compute $A_{\text{even}}(x^2)$ and $A_{\text{odd}}(x^2)$ for all the values x that are nth roots of unity, by first deleting every other column of V (which contain the odd powers of the nth roots of unity) and removing duplicate rows, and then multiplying by the vectors $\vec{A}_{\text{even}}$ and $\vec{A}_{\text{odd}}$ which consist of the components $a_0, a_2, a_4, \ldots, a_{n-1}$ and $a_1, a_3, a_5, \ldots, a_{n-1}$, respectively.

After the deletion of the columns, half the rows of V are duplicates since, when n is a power of 2, squaring the n distinct nth roots of unity gives us the $n/2$ distinct $(n/2)$th roots of unity. Assuming we've listed the nth roots of unity such that roots squaring to the same value are adjacent (as we have in our example), we can think of these multiplication as

$$\begin{pmatrix} 1 & x_0^2 & \ldots & x_0^{n-2} \\ 1 & x_2^2 & \ldots & x_2^{n-2} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n-4}^2 & \ldots & x_{n-4}^{n-2} \\ 1 & x_{n-2}^2 & \ldots & x_{n-2}^{n-2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_2 \\ \vdots \\ a_{n-2} \end{pmatrix} = \begin{pmatrix} A_{\text{even}}(x_0^2) \\ A_{\text{even}}(x_2^2) \\ \vdots \\ A_{\text{even}}(x_{n-4}^2) \\ A_{\text{even}}(x_{n-2}^2) \end{pmatrix}$$

and

$$\begin{pmatrix} 1 & x_0^2 & \ldots & x_0^{n-2} \\ 1 & x_2^2 & \ldots & x_2^{n-2} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n-4}^2 & \ldots & x_{n-4}^{n-2} \\ 1 & x_{n-2}^2 & \ldots & x_{n-2}^{n-2} \end{pmatrix} \begin{pmatrix} a_1 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} A_{\text{odd}}(x_0^2) \\ A_{\text{odd}}(x_2^2) \\ \vdots \\ A_{\text{odd}}(x_{n-4}^2) \\ A_{\text{odd}}(x_{n-2}^2) \end{pmatrix}.$$

For our example, these multiplications are

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \\ 1 & -i & -1 & i \end{pmatrix} \begin{pmatrix} 5 \\ -2 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 7 \\ 5-2i \\ 5+2i \end{pmatrix}$$

and

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \\ 1 & -i & -1 & i \end{pmatrix} \begin{pmatrix} 3 \\ 4 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 7 \\ -1 \\ 3+4i \\ 3-4i \end{pmatrix}.$$

Notice we are now performing two matrix-vector multiplications with a 4×4 matrix containing 4 distinct values.

This means that in our example,

$$\begin{aligned} A(-i) &= A_{\text{even}}((-i)^2) + (-i)A_{\text{odd}}((-i)^2) \\ &= A_{\text{even}}(-1) + (-i)A_{\text{odd}}(-1) \\ &= 7 + (-i)(-1) \\ &= 7 + i\,, \end{aligned}$$

for instance. (We can tell $A_{\text{even}}(-1) = 7$ and $A_{\text{odd}}(-1) = -1$ because the second components of the matrix-vector products immediately above are 7 and -1.) Checking the inner product of the 4th row of V — in which the components are the powers of $x_3 = -i$ — with the transpose of $\vec{A}$ in our example, indeed we have

$$(1, -i, -1, i, 1, -i, -1, i) \cdot (5, 3, -2, 4, 0, 0, 0, 0) = 5 - 3i + 2 + 4i + 0 + 0 + 0 + 0 = 7 + i\,.$$

Using this divide-and-conquer approach, the time to evaluate $A(x)$ on the nth roots of unity by computing $V\vec{A}$ is $T(n) = 2T(n/2) + O(n) = \Theta(n \log n)$. If we do the same for $B(x)$, and then compute $C(x_1) = A(x_0)B(x_1), \ldots, C(x_{n-1}) = A(x_{n-1})B(x_{n-1})$, we'll have n distinct points on the curve $C(x)$, which uniquely specify $C(x)$.

To recover the coefficients of $C(x)$ from $C(x_0), \ldots, C(x_{n-1})$, we multiply $V^{-1}\vec{C}$, where V^{-1} is V's inverse and $\vec{C}$ is the vector with components $C(x_0), \ldots, C(x_{n-1})$. We can do this with the Inverse FFT (IFFT), which also works in $O(n \log n)$ time and is based on the fact that $V^{-1} = \overline{V}/n$, where $\overline{V}$ is the result of replacing each entry of V by its complex conjugate (that is, multiplying each entry's imaginary part by -1).

Notice that taking the complex conjugates of the matrix in our example only swaps the rows. This is true in general, because taking complex conjugates doesn't change the set of nth roots of unity (and their powers), it just rearranges them. Therefore, we can compute the IFFT the same way we computed the FFT, and then multiply the resulting vector by $1/n$.

Assignment 2

posted: 14.05.2021 **due: midnight 28.05.2021**

You can work in groups of up to three people. One group member should submit a copy of the solutions on Brightspace, with all members' names and banner numbers on it; the other group members should submit text files with all members' names and banner numbers (otherwise Brightspace won't let us assign them marks!). You may consult with other people but each group should understand the solutions: after discussions with people outside the groups, discard any notes and do something unrelated for an hour before writing up your solutions; it's a problem if no one in a group can explain one of their answers. For programming questions you should submit your code, which should compile and run correctly to receive full marks.

1. Given a tree on n vertices, we can always find a single vertex whose removal leaves a forest in which no tree has more than $n/2$ vertices. Suppose we use our divide-and-conquer algorithm to count the 3-colourings of a tree on n vertices; about how long does it take? How fast can you compute the answer?
   You can assume n is a power of 2 and the tree is always split into exactly two pieces of size $n/2$ (even though the two pieces together should have $n - 1$ vertices instead of n, since we removed a vertex to split the tree).

2. In the lecture, we saw that implementing Euclid's algorithm on positive integers a and b with $a > b$ by repeated subtraction takes $\Omega(a)$ time in the worst case but implementing it by mod takes $O(\log a)$ time, *assuming subtraction and* mod *each take constant time.* Now suppose subtracting two n-digit numbers takes n time but taking their mod takes n^2 time; comparing two numbers takes time 1. About how much bigger does a have to be than b in order for it to be faster to compute $a \bmod b$ with mod directly than with repeated subtraction?
   For example, if $a = 1523$ and $b = 0427$, then computing $a \bmod b = 242$ by repeated subtraction means subtracting 0427 from 1523 to get 1096 in 4 time units, checking 1096 is still bigger than 0427 in 1 time unit, subtracting 0427 from 1096 to get 0669 in 4 time units, checking 0669 is still bigger than 0427 in 1 time unit, subtracting 0427 from 0667 to get 0242 in 4 time units, and checking whether 0240 is bigger than 0427 in 1 time unit (and finding it's not). That takes a total of $4 + 1 + 4 + 1 + 4 + 1 = 15$ time units, whereas computing $a \bmod b = 242$ directly takes $4^2 = 16$ time units, so in this case repeated subtraction is faster.

3. Describe how to build a circuit consisting of AND, OR and NOT gates that takes two n-bit binary numbers x and y and outputs the $(n+1)$-bit binary number $x+y$. Your circuit should be a directed acyclic graph (a DAG) whose size is at most polynomial in n and whose depth is constant (where "depth" means the length of the longest directed path); the fan-in and fan-out are not bounded (where "fan-in" and "fan-out" mean the maximum in- and out-degree of any vertex).

Continued on Next Page!

4. Give a divide-and-conquer program for

   `https://leetcode.com/problems/maximum-subarray`

   (you don't have to pay for a membership!) and explain how to use your solution to solve

   `https://leetcode.com/problems/maximum-sum-circular-subarray`

   neatly.

   (If you don't use divide-and-conquer or your solution looks like it's been copied, you will *not* get the mark and you may be reported to FCS.)

5. Suppose you didn't understand Max's lecture on the FFT but you still want to multiply degree-n polynomials in time subquadratic in n. Show how to use Karatsuba's algorithm to do it in $O(n^{\lg 3})$ time, assuming arithmetic operations on coefficients take constant time.
   For example, consider multiplying the two polynomials

$$\begin{aligned} A(x) &= 8x^7 + 5x^6 + 4x^5 + x^4 + 9x^3 + 6x^2 + 2x + 1 \\ B(x) &= 7x^7 + 5x^6 + 3x^5 + 3x^4 + 9x^3 + 4x^2 + 5\,. \end{aligned}$$

   Notice

$$\begin{aligned} &(8x^3 + 5x^2 + 4x + 1)(9x^3 + 4x^2 + 5) + (9x^3 + 6x^2 + 2x + 1)(7x^3 + 5x^2 + 3x + 3) \\ &= (8x^3 + 5x^2 + 4x + 1 + 9x^3 + 6x^2 + 2x + 1)(7x^3 + 5x^2 + 3x + 3 + 9x^3 + 4x^2 + 5) - \\ &\quad (8x^3 + 5x^2 + 4x + 1)(7x^3 + 5x^2 + 3x + 3) - (9x^3 + 6x^2 + 2x + 1)(9x^3 + 4x^2 + 5)\,; \end{aligned}$$

   does this look familiar?

Chapter 5

Asymptotic Bounds

My friend Simon was part of a team that took turns running on a treadmill and together ran over 420 km in 24 hours, trying to break the world record (about 429 km back then, currently about 453 km). His ambition was to be the first PhD-level computer scientist to run faster than Turing, although I don't think he quite made it.[*] Suppose Simon normally runs distance $n \geq 1$ in time $f(n) = n^{3/2}$. His exact performance on any given day depends on many factors, however: if he's wearing his favourite shoes then we should multiply his time by 0.8; if he's just had a coffee, we multiply it by 0.9; if he's tired, we multiply it by 1.1; etc.

Suppose I normally run distance $n \geq 1$ in time $n^{7/4}$, but if I'm running against Simon then he might give me a head start, so I'll take time $g(n) = n^{7/4} - 10$. If I'm wearing my favourite shoes then we should multiply my time by 0.95, etc. Clearly Simon is much faster than I am in the long run (pun intended), even though in a short enough race and with a big enough head start and when I'm wearing my favourite shoes and he's not, etc, I might still win. How can we formalize the notion of "in the long run" and our intuition that, regardless of head starts and shoes and coffee and fatigue, as long as we run far enough then Simon is going to leave me in the dust?

This is important in computer science because, when we're talking about how much time algorithms take, we don't want to worry too much about how they behave on small inputs, or whether one algorithm is slightly faster when implemented in some particular language on some particular architecture. For example, Simon is also a really good programmer and he works in Silicon Valley

[*]Incidentally, if you've seen *The Imitation Game* and think Benedict Cumberbatch's wimpy portrayal of Turing was accurate, think again: he sometimes ran more than 60 km from Bletchley Park to London for meetings. When he was 13 he was featured in a local newspaper for cycling over 100 km to his boarding school during a train strike:

> "By 1926, Alan's parents were living in France and with Alan due to begin his first term at Sherborne School on 4 May 1926 he set off alone the preceding day, taking the Channel ferry from St Malo and arriving in Southampton only to discover that owing to the General Strike no trains were running. Undeterred, Alan sent his housemaster a telegram informing him that he would not now be arriving until the following day and set out to cycle the 63 miles from Southampton to Sherborne across what was for him unknown territory. He stopped overnight at the Crown Hotel in Blandford Forum and the following morning, he reported later in a letter to his parents, the hotel staff turned out to see him off on the remainder of his journey to Sherborne."

(`https://oldshirburnian.org.uk/the-sherborne-formula-the-making-of-alan-turing`) so he can probably afford a faster computer than I can, but maybe his brain has atrophied slightly since he switched from academia to industry so he's not as good at designing algorithms any more. (I admit this is unlikely, but it's a convenient supposition for this lecture.) Let's suppose he designs an algorithm that can normally process an input of size n in $g(n)$ time and I design an algorithm for the same problem that can normally process such an input in $f(n)$ time. Even if Simon can speed his algorithm up by a factor of 1.5 using fancy programming tricks, and by another factor of 2 using a fancy computer, and his algorithm has a head start because he pre-computes what it should do for the first few steps on any input, in the long run my algorithm is still going to be faster.

One way to say what we want is

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0\,,$$

but computer scientists usually use asymptotic notation and just say "$f(n) \in o(g(n))$" or, equivalently, "$g(n) \in \omega(f(n))$". You've seen O in previous courses and we've mentioned Ω and Θ before in this course, and I told you to think of them as meaning "at most about", "at least about" and "about", respectively. (Historically, O stood for "on the order of", which is a bit confusing because in English that means "about", not "at most about".) In this class we'll cover o and ω too and give formal definitions, then talk about proving upper and lower bounds, and finally about how to use the Master Theorem to solve many of the recurrences you'll run across using divide-and-conquer algorithms.

The first thing we should get straight is that $O(g(n))$, $\Omega(g(n))$, $\Theta(g(n))$, $o(g(n))$ and $\omega(g(n))$ are all *sets* — more specifically, sets of functions — and we should write $f(n) \in O(g(n))$ instead of $f(n) = O(g(n))$, even though being picky about the proper use of asymptotic notation is as frustrating as being picky about the proper use of apostrophes. It's really common for people to write $f(n) = O(g(n))$, but it's also misleading, since equality is supposed to be reflexive — that is, if $x = y$ then $y = x$ — but $f(n) = O(g(n))$ (which is true in this case) doesn't imply $g(n) = O(f(n))$ (which is false in this case).

Being really precise, $O(g(n))$ is the set of functions $h(n)$ such that, for some constants n_0 and $c > 0$ and for every $n \geq n_0$, we have $h(n) \leq cg(n)$. We can write this in mathematical notation, with $\exists$ for "there exists" and $\forall$ for "for all", as

$$O(g(n)) = \{h(n) \,:\, \exists\, n_0, c > 0 \;\forall\, n \geq n_0 \,.\, h(n) \leq cg(n)\}\,.$$

Notice our $f(n)$ is in this set, $f(n) \in O(g(n))$. You can see that "about" in my "at most about" means "for sufficiently large n and ignoring constant factors".

To go from "at most about" to "at least about", we just switch things around:

$$\Omega(f(n)) = \{h(n) \,:\, \exists\, n_0, c \;\forall\, n \geq n_0 \,.\, h(n) \geq cf(n)\}\,.$$

I've switched from $g(n)$ to $f(n)$ here because, for the functions $f(n)$ and $g(n)$ we've chosen, we have $g(n) \in \Omega(f(n))$ but $f(n) \notin \Omega(g(n))$. Of course if we'd chosen $f(n) = n^{3/2}$ and $g(n) = 5n^{3/2} + \lg n - 7$, for example, then we'd have all of $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$ and $f(n) \in O(g(n))$ and $f(n) \in Omega(g(n))$. In this case, we write $f(n) \in \Theta(g(n))$ or $g(n) \in \Theta(f(n))$, so when I said Θ

means "about", I meant "within a positive constant factor of each other, for sufficiently large n". Formally, $\Theta(f(n)) = O(f(n)) \cap \Omega(g(n))$.

Notice $f(n) \in O(g(n))$ implies $g(n) \in \Omega(f(n))$ and vice versa, so $f(n) \notin O(g(n))$ implies $g(n) \notin \Omega(f(n))$ and vice versa, but it's *not* true that $n \sin n \notin O(n \cos n)$ implies $n \sin n \in \Omega(n \cos n)$, for example. Although for real numbers x and y either we have $x < y$ or $x = y$ or $x > y$, functions like $n \sin n$ and $n \cos n$ aren't always comparable like that, because for any constant c we have $(n \sin n)/(n \cos n) > c$ for infinitely many values of n, and $(n \cos n)/(n \sin n) > c$ for infinitely many values of n.

Of course, none of these concepts really say what we started out wanting to say about how Simon and I run: he's not just at least about as fast as I am, in the long run he's much faster. For that we need o or ω. Saying $f(n) \in o(g(n))$ means that for any positive constant c and sufficiently large n, we have $f(n) \leq c(n)$ or, in mathematical notation,

$$o(g(n)) = \{h(n) \,:\, \forall\, c > 0\ \exists\, n_0\,.\ h(n) \leq c g(n)\}\,.$$

Switching things around, saying $g(n) \in \omega(f(n))$ means that for any positive constant c and sufficiently large n, we have $g(n) \geq c f(n)$ or, in mathematical notation,

$$\omega(f(n)) = \{h(n) \,:\, \forall\, c > 0\ \exists\, n_0\,.\ h(n) \geq c f(n)\}\,.$$

Notice $f(n) \in o(g(n))$ implies $g(n) \in \omega(f(n))$ and vice versa, and they're both equivalent to saying

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

or

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \infty\,.$$

Since $O(g(n))$, $\Omega(g(n))$, $\Theta(g(n))$, $o(g(n))$ and $\omega(g(n))$ are all sets, we can draw them as a Venn diagram, with the following relations:

$$\begin{array}{rcl} o(g(n)) & \subset & O(g(n))\,, \\ \Theta(g(n)) & = & O(g(n)) \cap \Omega(g(n))\,, \\ \omega(g(n)) & \subset & \Omega(g(n))\,, \\ g(n) & \in & \Theta(g(n))\,, \\ g(n) & \notin & o(g(n)) \cup \omega(g(n))\,, \end{array}$$

as shown in Figure 5.1. Notice we can write $\subset$ instead of $\subseteq$, because $g(n) \in \Theta(g(n))$ but $g(n) \notin o(g(n)) \cup \omega(g(n))$.

In fact, since $f(n) \in o(g(n))$, we also have the following relations, which are beyond my abilities at drawing Venn diagrams:

$$\begin{array}{rcl} o(f(n)) & \subset & O(f(n))\,, \\ \Theta(f(n)) & = & O(f(n)) \cap \Omega(f(n))\,, \end{array}$$

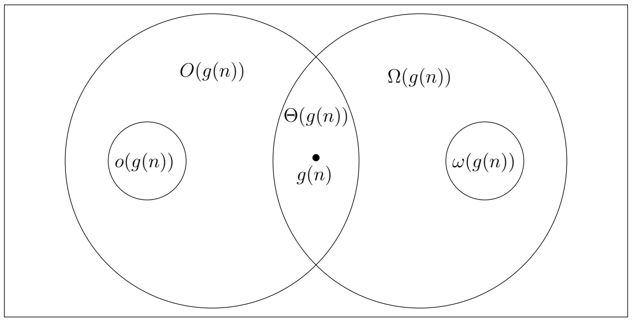


Figure 5.1: A Venn diagram showing the relationships between $O(g(n))$, $\Omega(g(n))$, $\Theta(g(n))$, $o(g(n))$ and $\omega(g(n))$.

$$\begin{aligned}
\omega(f(n)) &\subset \Omega(f(n))\,, \\
f(n) &\in \Theta(f(n))\,, \\
f(n) &\notin o(f(n)) \cup \omega(f(n))\,, \\
O(f(n)) &\subset o(g(n))\,, \\
\Omega(g(n)) &\subset \omega(f(n))\,.
\end{aligned}$$

It's traditional for instructors to give their students lots of examples of functions and how they compare asymptotically, and drive home the message of how asymptotic notation lets us ignore constant factors and lower-order terms. Because that's traditional, however, you can find lots of explanations and exercises on Khan Academy or YouTube, as well as in *Introduction to Algorithms*. Actually, one of the coauthors of the lesson on Khan Academy (`https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation`) is a coauthor of *Introduction to Algorithms*.

Instead, I want to focus on something I've heard from students in the past, which is that Ω is for best-case bounds and O is for worst-case bounds. I don't know where people get that from, but it's WRONG!

Stirling's formula says
$$\ln n! = n \ln n - n + O(\log n)\,.$$
(The sum of the lower-order terms hidden by $O(\log n)$ is positive. If you want to be really exact, it's between $(1/2)\ln n + \ln\sqrt{2\pi} + 1/(12n+1)$ and $(1/2)\ln n + \ln\sqrt{2\pi} + 1/(12n)$.) Because it includes an O, is Stirling's formula talking about the worst case? In what sense is it bad? Incidentally, notice that we don't care about the base of the logarithm, since changing from one constant greater than one to another just changes the constant coefficient hidden by the O.

It often makes sense to say things like "even in the best case, we still use at least about $\sqrt{n}$ time" or "even in the worst case, we still use at most about n^2 time", and then we indeed use Ω for a best-case bound and O for a worst-case bound, but it can also make sense to say "in the best case we use *at most* about $n^{3/4}$ time" or "in the worst case we use *at least* about $n \log n$ time".

$$\begin{matrix}
1 & 1 & 1 & 1 & 1 & 3 \\
1 & 1 & 1 & 1 & 3 & 4 \\
1 & 1 & 1 & 3 & 4 & 4 \\
1 & 1 & 3 & 4 & 4 & 4 \\
1 & 2 & 4 & 4 & 4 & 4 \\
3 & 4 & 4 & 4 & 4 & 4
\end{matrix}$$

Figure 5.2: A 6×6 matrix with 1s in the top left, 4s in the bottom right and a diagonal of 3s separating them, except for a single 2.

Instead of associating Ω with best-case bounds and O with worst-case bounds, it's more accurate to say Ω is for lower bounds (or "bounds from below") and O is for upper bounds (or "bounds from above"). Let's look at my favourite example.

Suppose I give you a scratchcard with an $n \times n$ matrix on it, and tell you that in each cell there is a number between 1 and n^2, arranged such that every row and every column is sorted in non-decreasing order from left to right and from top to bottom. You draw a target number x from a hat with n^2 pieces of paper, one with each number from 1 to n^2, and then have to scratch enough cells to either find x in the matrix or show it isn't there. How many cells do you need to scratch?

Even in the best case, you obviously need to scratch at least one cell, and if you're lucky and draw $x = 1$, then you just need to check the top left, so the best-case bound is $\Omega(1) \cap O(1) = \Theta(1)$. When our lower bound and upper bound match for this case, ignoring constant coefficients and lower-order terms (although those match too in this instance), we can use Θ and we say the bound is *tight*.

What about in the worst case? Clearly, you still need $\Omega(1)$ in the worst case, and you don't have to scratch more than the whole matrix, so $O(n^2)$ is enough — but we don't have a tight bound yet. If you perform binary search on each row, then you scratch $O(n \log n)$ cells. It feels like you might be able to use some kind of 2-dimensional binary search, first checking the cell in the middle of the matrix and eliminating either the top-left quadrant or the bottom-right quadrant. Does that work?

In fact it doesn't. To see why, suppose the top left of the matrix is full of 1s, the bottom right is full of 4s, and the diagonal separating them is a line of 3s, except one of the cells on the diagonal contains a 2, as shown in Figure 5.2. If you're unlucky and draw an $x = 2$, then you have to check at least every cell on the diagonal until you find the 2, because if you don't then you can't be sure if there's a 2 or not. Whatever your strategy, the 2 could be in the last cell you check, so in the worst case you need to scratch $\Omega(n)$ cells. That is, tempting though it is, binary search won't help much in the worst case.

It's not hard to see that we can take the idea we just used to prove a lower bound on the number of cells you need to scratch, and derive a matching upper bound. Suppose we start at the top-left corner and move right, scratching each cell as we go, until we either find x (in which case you're done) or a value greater than x or we reach the top right-corner, in which case we turn and go down and continue until we find x or a value greater than x or we reach the bottom-right corner (in which case you know x is not in the matrix). Assuming we find a value greater than x, you can trace out the border between the values less than x and at least x by scratching $O(n)$ cells. Since x has to lie along that border, you can find it or determine that it's not in the matrix, scratching only those $O(n)$ cells. Since our upper bound $O(n)$ matches our lower bound $\Omega(n)$, we have a tight bound, $\Theta(n)$.

Now that we more or less understand asymptotic notation (it gets more complicated with more than one variable, but we won't worry about that in this course), we can understand what the Master Theorem says. There are stronger versions, but the one stated in *Introduction to Algorithms* says this:

Theorem 1 *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence*

$$T(n) = aT(n/b) + f(n)\,,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.*
2. *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.*
3. *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

This won't help you solve all recurrences, nor even all recurrences you'll encounter with divide-and-conquer algorithms — for example, the recurrence $T(n) = 3^{2\sqrt{n}} \cdot 2T(2n/3)$ from Chapter 2 doesn't have the right form, since $3^{2\sqrt{n}} \cdot 2$ isn't a constant — but it will help with a lot of them. For example, it will help with the recurrences for Karatsuba's algorithm, $T(n) = 3T(n/2) + 2n$, and for Strassen's algorithm, $T(n) = 7T(n/2) + cn^2$ for some constant c.

For the recurrence for Karatsuba's algorithm, we have $a = 3$, $b = 2$ and $f(n) = 2n$, and $2n = O(n^{\log_2 3 - \epsilon})$ for $\epsilon < 0.58$, so we're in Case 1 and $T(n) = \Theta(n^{\lg 3})$. For the recurrence for Strassen's algorithm, we have $a = 7$, $b = 2$ and $f(n) = cn^2$ for some constant c, and $cn^2 = O(n^{\log_2 7 - \epsilon})$ for $\epsilon < 0.8$, so we're again in Case 1 and $T(n) = \Theta(n^{\lg 7})$. You've probably seen the recurrence for MergeSort, $T(n) = 2T(n/2) + cn$, and for that we're in Case 2, since $a = 2$, $b = 2$ and $cn = \Theta(n^{\log_2 2}) = \Theta(n)$, so MergeSort takes $\Theta(n \log n)$ time.

If we think of the trace of a divide-and-conquer algorithm as a tree of subproblems, then we can give informal characterizations of the three cases. Case 1 is when we're subdividing the problem into many subproblems, but the subproblems aren't shrinking as fast as we're subdividing. For example, with Strassen's algorithm we're dividing 2 matrices of size $n \times n$ into a total of 8 submatrices and then doing 7 multiplications, but the submatrices are still of size $(n/2) \times (n/2)$. This means that most of the work in the tree is done at the leaves.

Case 2 is when we're subdividing and the subproblems are shrinking at about the same speed, so we do about the same amount of work at each level of the tree. For example, in MergeSort, in the ith round of the recurrence, we're doing linear work on 2^i lists each of length about $n/2^i$, which takes $O(n)$ time in total for that round.

Case 3 is when the overhead of splitting the problem into subproblems and then merging the subsolutions into a solution is really high, so most of the work is at the root of the tree. For example, suppose it takes us 2^n time to split the problem up into two subproblems each of size $n/2$ and later to merge the subsolutions of those into a solution for our original problem. Then the recurrence is

$$T(n) = 2T(n/2) + 2^n$$

and we have $a = 2$, $b = 2$ and $f(n) = 2^n = \Omega(n^{\log_2 2 + \epsilon})$ for any constant ϵ, so $T(n) = \Theta(2^n)$. This makes sense because

$$\begin{aligned}
T(n) &= 2T(n/2) + 2^n \\
&= 2(2T(n/4) + 2^{n/2}) + 2^n \\
&= 2(2(2T(n/8) + 2^{n/4}) + 2^{n/2}) + 2^n \\
&\vdots \\
&= 2^{\lg n} + \sum_{i=0}^{\lg n - 1} 2^{n/2^i + i}
\end{aligned}$$

is dominated by the term 2^n.

Chapter 6

Sorting

You may have heard that sorting n integers takes $O(n \log n)$ time. As long as each integer fits in a constant number of machine words, that's true, as we'll see in this lecture. In fact, we'll see that sorting n integers takes $\Theta(n \log n)$ time in the worst case in some models (and by now you should understand why saying $\Theta(n \log n)$ is stronger than saying $O(n \log n)$). Before we consider cases when we actually need $\Omega(n \log n)$ time, though, we should discuss two special cases when it's possible to do better, the latter of which comes up fairly often in practice.[*]

Suppose we have n positive binary integers $x_1, \ldots, x_n$ drawn from a range of size cn for some constant c, with each x_i fitting in a constant number of machine words. We can easily scan them to find $\min_j\{x_j\}$ and then subtract it from each x_i to get n integers $x'_1, \ldots, x'_n$ in the range $[0, cn-1]$, all in $O(n)$ time. If we can sort $x'_1, \ldots, x'_n$ in $O(n)$ time, then we can just add $\min_j\{x_j\}$ to each x'_i and recover $x_1, \ldots, x_n$, sorted. Therefore, without loss of generality, we can assume $x_1, \ldots, x_n$ are from the range $[0, cn-1]$.

In $O(cn) = O(n)$ time we create an integer array $A[0..cn-1]$ with each $A[i] = 0$ initially. We scan $x_1, \ldots, x_n$ and, for each x_i, we increment $A[i]$. Finally, we scan $A[0..cn-1]$ and, for each $A[i]$, we report that there were that many occurrences of i in $x_1, \ldots, x_n$. This all takes $O(n)$ time, and the procedure is called *counting sort*. If we want to return the actual x_is (perhaps because they have satellite data associated with them), then we can make A an array of linked lists instead, and add each x_i to the appropriate linked list instead of just incrementing the appropriate counter. Notice that, if we prepend each x_i to the head of its linked list but eventually report the contents of each linked list in tail-to-head order (which still takes linear time), then we sort $x_1, \ldots, x_n$ *stably*: that is, if $x_i = x_j$ and $i < j$, then we report x_i before x_j.

Now suppose we have n positive binary integers $x_1, \ldots, x_n$ drawn from a range of size n^c for some constant c, with each x_i still fitting in a constant number of machine words. Again, without loss of generality, we can assume $x_1, \ldots, x_n$ are from the range $[0, n^c - 1]$. Let $b = \lceil \lg n \rceil$, so we can view each x_i as a sequence of c binary numbers, each of b bits and in the range $[0, 2^{\lceil \lg n \rceil}]$. Since

[*]In the word-RAM model, which is our default for most of this course, I think the best known upper bound for sorting n integers that each fit in a constant number of machine words, is $O(n\sqrt{\log \log n})$, by Yan and Thorup from FOCS 2002; we won't discuss that in this course, though.

$2^{\lceil \lg n \rceil} < 2n$, we can then view $x_1, \ldots, x_n$ as an $n \times c$ matrix, with each cell containing a number in the range $[0, 2n - 1]$.

This is like viewing each x_i as a c-digit number in base 2^b, and we can use *radix sort* to sort $x_1, \ldots, x_n$ quickly. For each of the c columns, we use counting sort to sort in $O(n)$ time the rows of our matrix on the basis of the numbers in that column. (Swapping two rows takes constant time, because each x_i fits in a constant number of machine words.) Overall we use $O(cn) = O(n)$ time so, as long as we're working in the word-RAM model and each integer fits in a constant number of machine words, we can sort n integers from a range of size polynomial in n in linear time.

If we sort based on the the columns of the matrix working from left to right, it corresponds to most-significant-bit-first (MSB) radix sort (although our "bits" in this case are digits in base 2^b); if we work right to left, it corresponds to least-significant-bit-first (LSB) radix sort. Each of these have advantages and disadvantages: for example, with MSB, if we get interrupted halfway through, we've still roughly sorted $x_1, \ldots, x_n$; with LSB, we don't need to split the matrix into buckets as long as we use stable counting sort.

To see what I mean, consider the example

$$\begin{aligned}
x_1 &= 10001101\,,\\
x_2 &= 11101001\,,\\
x_3 &= 00110010\,,\\
x_4 &= 01001000\,,\\
x_5 &= 01001010\,,\\
x_6 &= 10100101\,,\\
x_7 &= 01110100\,.
\end{aligned}$$

(Let's skip subtracting the minimum this time.) Since $n = 7$ we have $b = 2^{\lceil \lg n \rceil} = 8$ and we view each x_i as a 3-digit octal number (or, equivalently, we view $x_1, \ldots, x_n$ as a matrix with 7 rows and 3 columns with each cell containing a number between 0 and 7):

$$\begin{aligned}
x_1 &= 215\,,\\
x_2 &= 351\,,\\
x_3 &= 062\,,\\
x_4 &= 110\,,\\
x_5 &= 112\,,\\
x_6 &= 245\,,\\
x_7 &= 164\,.
\end{aligned}$$

If we use counting sort on the least significant digits, we get

$$\begin{aligned}
x_4 &= 110\,,\\
x_2 &= 351\,,\\
x_3 &= 062\,,\\
x_5 &= 112\,,
\end{aligned}$$

$$\begin{aligned} x_7 &= 164\,, \\ x_1 &= 215\,, \\ x_6 &= 245\,. \end{aligned}$$

Now if we use counting sort on the second digits, ignoring the least significant ones, if we're not careful we *could* get

$$\begin{aligned} x_1 &= 215\,, \\ x_5 &= 112\,, \\ x_4 &= 110\,, \\ x_6 &= 245\,, \\ x_2 &= 351\,, \\ x_7 &= 164\,, \\ x_3 &= 062\,, \end{aligned}$$

which undoes all the sorting we just did based on the least significant digits. If we use *stable* counting sort on the second digits, however, then we get

$$\begin{aligned} x_4 &= 110\,, \\ x_5 &= 112\,, \\ x_1 &= 215\,, \\ x_6 &= 245\,, \\ x_2 &= 351\,, \\ x_3 &= 062\,, \\ x_7 &= 164\,, \end{aligned}$$

with x_4 preceding x_5, for example, because it preceded it after we sorted on the least significant digit, because 0 is smaller than 2 and those are x_4's and x_5's least significant digits, respectively. Applying stable counting sort to the most significant digits, we get

$$\begin{aligned} x_3 &= 062\,, \\ x_4 &= 110\,, \\ x_5 &= 112\,, \\ x_7 &= 164\,, \\ x_1 &= 215\,, \\ x_6 &= 245\,, \\ x_2 &= 351\,. \end{aligned}$$

Finally, translating back into binary, we have

$$\begin{aligned} x_3 &= 000110010\,, \\ x_4 &= 001001000\,, \end{aligned}$$

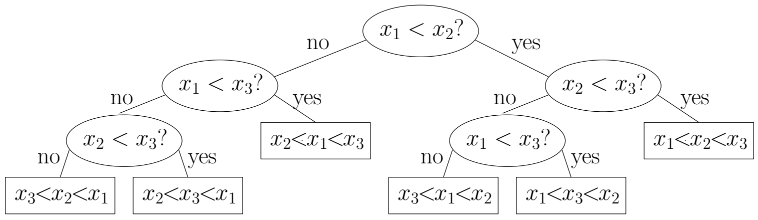


Figure 6.1: A decision tree for sorting 3 distinct numbers.

$$\begin{aligned}
x_5 &= 001001010\,,\\
x_7 &= 001110100\,,\\
x_1 &= 010001101\,,\\
x_6 &= 010100101\,,\\
x_2 &= 011101001\,.
\end{aligned}$$

I guess you might have seen all this in first or second year, but it's probably good to have a reminder before continuing, anyway. Just in case, I'm going to restate what we saw above as a theorem:

Theorem 2 *In the word-RAM model, if each integer fits in a constant number of machine words then sorting n integers from a range of size polynomial in n takes $O(n)$ time.*

Counting sort and radix sort "look inside" their arguments, but for the rest of the lecture we'll be working in the *comparison model*, which disallows direct addressing, pointer arithmetic, and even adding and subtracting the integers in the input. Since the word-RAM model lets us do more (as long as the integers each fit in a constant number of machine words so we can compare two of them in constant time), upper bounds in the comparison model hold in the word-RAM model but not necessarily vice versa,[†] and lower bounds in the word-RAM model hold in the comparison model but not necessarily vice versa. This means showing that MergeSort works in $O(n \log n)$ time in the comparison model, for example, implies we can sort in that time in the word-RAM model as well.

First, let's see why we need $\Omega(n \log n)$ time in the worst case when we're in the comparison model. Suppose we have a set of n numbers, $x_1, \ldots, x_n$, now with "set" implying they're all distinct. A pairwise comparison between x_i and x_j returns exactly 1 bit of information: either $x_i < x_j$ or $x_i > x_j$. Suppose you have an algorithm A that does pairwise comparisons and eventually tells you how to order $x_1, \ldots, x_n$, and consider A as a decision tree. For example, Figure 6.1 shows a decision tree for sorting 3 numbers with pairwise comparisons.

[†]Sort of: some people only count comparisons in the comparison model and allow other computations for free, in which case those bounds may not translate well into the RAM model.

Since the decision tree is binary and has a leaf for each of the $n!$ possible orderings, one of those leaves must be at depth at least $\lceil \lg n! \rceil$. Stirling's Formula says that's

$$\left\lceil \frac{\ln n!}{\ln 2} \right\rceil = \frac{1}{\ln 2}(n \ln n - n + O(\log n) = n \lg n - \frac{n}{\ln 2} + O(\log n) = \Omega(n \log n)\,.$$

Therefore, for some ordering, algorithm A makes $\Omega(n \log n)$ pairwise comparisons — regardless of what algorithm A is.

Lemma 3 *In the comparison model, sorting takes $\Omega(n \log n)$ time.*

Does this mean that we *always* need $\Omega(n \log n)$ time to sort n numbers? Consider how InsertionSort behaves on the list $1, 2, 3, \ldots, n$: after $i-1$ steps, it has removed $1, \ldots, i-1$ from its input list and inserted them into its output list; during step i, it removes i from the front of its input list and scans through its output list, comparing i to $1, \ldots, i-1$ in turn, and then finally adds i at the end of that list. In total, it uses $1 + 2 + 3 + \ldots + n - 1 = \Omega(n^2)$ comparisons. This is actually its worst case.

Now, however, consider how InsertionSort behaves on the list $n, n-1, n-2, \ldots, 3, 2, 1$: after $i-1$ steps, it has removed $n, n-1, \ldots, n-i+1$ from its input list and inserted them into its output list; during step i, it removes $n-i$ from the front of its input list, compares it to $n-i+1$ at the front of its output list, and adds i at the start of that list. In total, it uses $O(n)$ comparisons. This is actually its best case.

Most algorithms do pretty well on their best case — which is one reason comparing best-case performance isn't very informative — but not all. For example, SelectionSort always uses $\Theta(n^2)$ comparisons. It's more interesting to talk about the average case or expected case, which we'll see when we get to QuickSort. My friend Jérémy has spent *years* studying how sorting algorithms can take advantage of pre-sortedness in their inputs, such as being made up of only a few interleave increasing subsequences. I might be able to squeeze that into the next assignment.

I think you all know MergeSort, but let's review it quickly: given an array of $n \geq 2$ numbers, we divide it roughly in half and recursively sort both halves, then merge the sorted subarrays. To merge two sorted subarrays S_1 and S_2 of length n_1 and n_2 into a sorted array S of length $n = n_1 + n_2$, we can conceptually append ∞ to both lists and run the following code:

```
i = 0;
j = 0;
k = 0;
while (S1 [i] != infty || S2 [j] != infty) {
    if (S1 [i] < S2 [j]) {
        S [k] = S1 [i];
        i++;
        k++;
    } else {
        S [k] = S2 [j];
```

```
            j++;
            k++;
        }
    }
```

This takes time $O(n_1 + n_2) = O(n)$. Incidentally, is this implementation of MergeSort stable?

The recurrence for MergeSort is $T(n) = 2T(n/2) + n$, since splitting the input in half takes constant time (and no comparisons) and merging the sorted sublists takes n time and comparisons. This is Case 2 of the Master Theorem, $a = 2$, $b = 2$, $f(n) = n = Theta(n^{\log_b a}) = \Theta(n)$, so we use $\Theta(n \log n)$ time overall.

Lemma 4 *In the comparison model, sorting takes $O(n \log n)$ time.*

Notice that, even though we just proved that MergeSort uses $\Theta(n \log n)$ time, that only gives us an $O(n \log n)$ upper bound for sorting, until we combine it with Lemma 3 to get a tight bound:

Theorem 5 *In the comparison model, sorting takes $\Theta(n \log n)$ time.*

So, why not just stop here? Why are there literally dozens of other sorting algorithms? Well, some of them have other useful properties, like being adaptive to various kinds of pre-sortedness, and some are just faster in practice, such as QuickSort.

I think you also all know QuickSort, but we should probably review that too: given an array $S[1..n]$ of n numbers, we choose a pivot $S[i]$ somehow; scan S and then partition it into numbers smaller than $S[i]$, numbers equal to $S[i]$, and numbers greater than $S[i]$; and recursively sort the numbers smaller than $S[i]$ and the numbers bigger than $S[i]$.

How long can this take in the worst case? If we're extremely unlucky, S could contain the numbers 1 to n and we could choose as pivots $1, 2, 3, \ldots, n$. In this case, for the ith step we choose i as a pivot and spend $n - i$ time scanning $i + 1, \ldots, n$ and finding they're all bigger than i. Thus, in the worst case we use $(n - 1) + (n - 2) + \cdots + 3 + 2 + 1 = \Theta(n^2)$ time.

To see why anyone uses QuickSort, we must analyze how fast it *usually* runs. Suppose we pick a pivot $S[i]$ uniformly at random from S; what is the probability that neither the partition of numbers smaller than $S[i]$ nor the partition of numbers bigger than $S[i]$ contain more than $3n/4$ numbers? If we imagine S already sorted then, because we pick $S[i]$ uniformly at random, the probability it is in the first quarter — in which case the partition of numbers bigger than $S[i]$ can contain more than $3n/4$ numbers — is $1/4$, and the probability it is in the last quarter — in which case the partition of numbers smaller than $S[i]$ can contain more than $3n/4$ numbers — is also $1/4$, but the probability it is in neither of those quarters — in which case neither of those partitions has size more than $3n/4$ — is $1/2$. Imagining S sorted is just for analysis, of course; we don't really want to sort S before choosing $S[i]$ (although later we'll discuss an idea along the same lines).

Suppose we change our implementation of QuickSort to be fussier: if at some step it picks a pivot and scans and partitions the subarray it's trying to sort recursively, and finds that either the partition of numbers smaller than the pivot contains more than 3/4 of all the numbers in the subarray, or the partition of numbers bigger than pivot does, then it undoes the partitioning, picks another pivot and tries again. How many tries do we expect it to make before it finds a pivot it likes?

As we just argued, it makes exactly 1 try with probability $p \geq 1/2$; it makes exactly 2 tries with probability $p(1-p)$; it makes exactly 3 tries with probability $p(1-p)^2$; etc. Therefore, the expected number of tries is

$$\sum_{i \geq 1} ip(1-p)^{i-1} \leq \sum_{i \geq 1} i/2^i = 2\,,$$

and so the expected time before it recurses is linear in the size of the subarray.

As we did when considering the cases of the Master Theorem, let's visualize the recursion as a tree of height $O(\log_{4/3} n) = O(\log n)$, with each vertex corresponding to a subarray, each of the n leaves corresponding to a single number, and the size of each vertex's subarray being the number of leaves in its subtree. At each vertex — that is, at each step in the recursion — we expect to spend time linear in the size of the subarray, which means each leaf in its subtree is contributing constant expected time to that vertex. By linearity of expectation — that is, because the expected value of a sum is the sum of the expected values of the terms — the total expected time is proportional to the sum of the leaves' depths, or $O(n \log n)$. As long as you believe making QuickSort fussier doesn't actually speed it up (it doesn't), then we can conclude that QuickSort uses $O(n \log n)$ time in the expected case.

Since we're talking about QuickSort, we can quickly cover a related algorithm, called QuickSelect. Suppose we just want to find the kth number in sorted order in an array $S[1..n]$. Of course we could sort S to find the kth number in sorted order, but that seems like overkill. Suppose we run QuickSort, but we discard whichever partition doesn't contain the kth number. For example, suppose we're looking for the 197th number in an array of 300 numbers. If we choose a pivot and partition and it turns out there are 103 numbers smaller than the pivot and 3 numbers equal to the pivot, then we can discard those and recursively find the 91st number in the 294 numbers larger than the pivot.

With QuickSelect, we're conceptually just descending one branch of the recursion tree, which takes expected time at most proportional to

$$n + (3/4)n + (3/4)^2 n + \cdots + 1 = O(n)$$

although, like QuickSort, in the worst case QuickSelect uses $\Theta(n^2)$ time.

In fact, it's possible to find the kth largest number in linear time even in the worst case. To do this, we partition S into quintuples and find the median of each quintuple, which takes $O(n)$ time. We collect the medians into a set S' of size about $n/5$ and recursively find the median of S'. The median of S' needn't be the median of S, but it can't be in the first quarter or the last quarter: half the $n/5$ quintuples had medians smaller than the median of S' (ignoring rounding), and each of those quintuples had 3 numbers (the median of the quintuple and the two smaller elements) smaller than the median of S', so there are at least $3n/10$ numbers in S smaller than the median of S' and, symmetrically, there are at least $3n/10$ numbers in S larger than the median of S'. It follows that the median of S' is a good pivot.

Analyzing this algorithm is a bit complicated, because we're finding the pivot recursively and then recursing on one of the partitions, but the recurrence and its solution is given in *Introduction to Algorithms*.

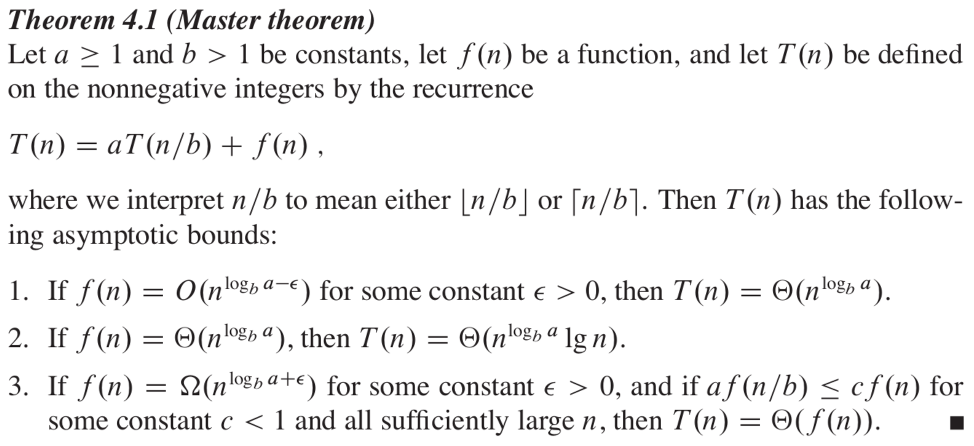
So, why don't people use this technique to find the pivots for QuickSort, to make it run in $O(n \log n)$ time in the worst case, as well as the expected case? Because then it's slower in practice than just running MergeSort!

Assignment 3

posted: 21.05.2021 **due: midnight 04.06.2021**

You can work in groups of up to three people. One group member should submit a copy of the solutions on Brightspace, with all members' names and banner numbers on it; the other group members should submit text files with all members' names and banner numbers (otherwise Brightspace won't let us assign them marks!). You may consult with other people but each group should understand the solutions: after discussions with people outside the groups, discard any notes and do something unrelated for an hour before writing up your solutions; it's a problem if no one in a group can explain one of their answers. For programming questions you should submit your code, which should compile and run correctly to receive full marks.

1. Mark the true statments. (Your score for this questions will be proportional to the number of statements you mark correctly, plus the number you correctly leave unmarked, minus the number you should have marked but didn't, minus the number you shouldn't have marked but did.)
   (a) $\lg n^n = O(\lg n!)$
   (b) $n^{(n+1) \bmod 2} = \Omega(n^{n \bmod 2})$ for $n \in \mathbb{N}$
   (c) $n^{(3n) \bmod 2} = O(n^{n \bmod 2})$ for $n \in \mathbb{N}$
   (d) If $T(n) = 7T(n/3) + n^4$ then $T(n) = \Omega(n^4/\log n)$.
   (e) If $T(n) = 8T(n/2) + 8n^3$ then $T(n) = \Theta(n^3 \log n)$.



Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$T(n) = aT(n/b) + f(n)$,

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

2. Explain how you can sort a sequence of n integers from a range of size $n^{\lg \lg n}$ in $O(n \log \log n)$ time (assuming each integer fits in a constant number of machine words).

3. An *in-place* algorithm uses a constant number of machine words of memory on top of the memory initially occupied by its input and eventually occupied by its output. Give code or pseudo-code for an in-place version of QuickSort.

Continued on Next Page!

4. You've probably seen in previous courses how to build a min-heap on n elements in $O(n)$ time and how to extract the minimum value from one in $O(\log n)$ time. Do you think we can easily extract the minimum value in $o(\log n)$ time while still leaving a heap on the remaining elements? Why or why not?

5. Suppose you have an algorithm that, given a sequence of n integers that can be partitioned into d non-decreasing subsequences but not fewer, does so in $O(n \log d)$ time. Explain how you can also sort such a sequence in $O(n \log d)$ time.

Part II

Greedy Algorithms

Chapter 7

Huffman Coding

When I asked during the first lecture for examples of profound truths we have found through computer science, someone suggested Shannon's information theory. I hadn't thought of that and it's a good answer, even though I think information theory is often considered more part of electrical engineering than computer science (for example, the *Transactions on Information Theory* is published by the Information Theory Society of the Institute of Electrical and Electronics Engineers, better known as the IEEE). I want to introduce greedy algorithms by teaching you Huffman's algorithm, which gives us a chance to discuss some of the basics of noiseless coding, which was the topic of the first half of Shannon's 1948 article introducing information theory (the other half was about noisy coding).

Before Shannon's article, it wasn't clear how to measure information. All other things being equal, it was intuitively clear that a big book contained more information than a one-page letter, but it was also easy to think the situation might be reversed if the book was boring and inaccurate while the letter held vitally important news. Shannon neatly sidestepped this by focusing only on transmitting *losslessly* whatever message we're given, regardless of its content. In fact, he assumed that we and the people we're transmitting to all know the probability distribution — over the universe of possible messages — according to which the message is chosen, and we've been able to prepare in advance a code assigning a distinct self-delimiting codeword to each possible message (where "possible" means the distribution assigns it a non-zero probability). This way, we don't have to talk about the message, per se, only about the distribution.

Shannon's information theory is thus concerned with *random variables* that take on values which are individual messages, not with the individual messages themselves. (Later in the course we'll talk a little about Kolmogorov complexity, which *is* concerned with the individual messages — but, as we'll see, that's incomputable.) We need not care about the significance of the random variables: a random variable that takes on the value "yes" with probability 0.5 and "no" with probability 0.5 is the same for us whether the question was "Launch the missile?" or "Do you want fries with that?". (My first statistics professor used to say "let's consider a random variable, such as your mark on the next test", which made us uncomfortable even though he was completely correct.) Some people still object to how information theory separates "information" from "meaning" and says a page of randomly-chosen characters contains more information than a page of beautiful poetry — but consider which is easier to memorize *exactly*.

One way to remember that information theory is about random variables is to think of transmitting information as removing uncertainty. It's not often mentioned in electrical engineering, but Shannon proposed three axioms for measuring uncertainty:

> Suppose we have a set of possible events whose probabilities of occurrence are $p_1, p_2, \ldots p_n$. These probabilities are known but that is all we know concerning which event will occur. Can we find a measure of how much "choice" is involved in the selection of the event or of how uncertain we are of the outcome?
>
> If there is such a measure, say $H(p_1, p_2, \ldots, p_n)$, it is reasonable to require of it the following properties:
>
> 1. H should be continuous in the p_i.
> 2. If all the p_i are equal, $p_i = \frac{1}{n}$, then H should be a monotonic increasing function of n. With equally likely events there is more choice, or uncertainty, when there are more possible events.
> 3. If a choice be broken down into two successive choices, the original H should be the weighted sum of the individual values of H. The meaning of this is illustrated in Figure 7.1. At the left we have three possibilities $p_1 = \frac{1}{2}, p_2 = \frac{1}{3}, p_3 = \frac{1}{6}$. On the right we first choose between two possibilities each with probability $\frac{1}{2}$, and if the second occurs make another choice with probabilities $\frac{2}{3}, \frac{1}{3}$. The final results have the same probabilities as before. We require, in this special case, that
>
> $$H\left(\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\right) = H\left(\frac{1}{2}, \frac{1}{2}\right) + \frac{1}{2} H\left(\frac{2}{3}, \frac{1}{3}\right) .$$
>
> The coefficient $\frac{1}{2}$ is because this second choice only occurs half the time.

Shannon showed that the only function H satisfying those three axioms is of the form

$$H = -K \sum_{i=1}^{n} p_i \log p_i \, ,$$

where K is a positive constant. Changing the base of the logarithm changes the value K or the unit in which we measure information and uncertainty. If the base of the logarithm is 2 and $K = 1$ then we are measuring uncertainty in bits, with 1 bit (an abbreviation of "binary digit" and a term suggested to Shannon by Tukey, one of the re-discoverers of the Fast Fourier Transform) being our uncertainty about the outcome of a flip of a fair coin. If the base of the logarithm is e and $K = 1$ then we are measuring uncertainty in units called nats (which I think only electrical engineers use).

He called this function the *entropy* of the random variable (or of the probability distribution, when we're concentrating on that). In this course, we'll usually be concerned with the binary entropy,

$$H(p_1, \ldots, p_n) = -\sum_{i=1}^{n} p_i \lg p_i \, ,$$

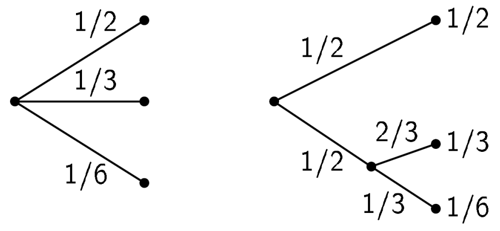


Figure 7.1: Decomposition of a choice from three possibilities (from Shannon's article).

and we'll measure information and uncertainty in bits. Quite apart from his axiomatic derivation he showed that, given $P = p_1, \ldots, p_n$, it is *not* possible to assign self-delimiting binary codewords to the n possible outcomes such that the expected codeword length is less than $H(P)$, but it is possible to assign such codewords so that the expected codeword length is less than $H(P)+1$. This result is often called "Shannon's Noiseless Coding Theorem".

It's not hard to prove the first part of the Noiseless Coding Theorem (the lower bound) but it's less important for this class, so we'll leave it until the end and for now prove only the second part (the upper bound).

Without loss of generality, suppose $p_1 \geq p_2 \geq \cdots \geq p_n > 0$. Let $s_i = \sum_{j=1}^{i-1} p_j$ and consider how many bits of the binary representation of s_i we must write to uniquely distinguish s_i. Notice that if two binary fractions agree on their first b bits after the binary point then they differ by less than $1/2^b$: the smaller binary fraction can end with an infinite number of 0s after those b bits, but the larger cannot end with an infinite number of 1s because we would write $0.0101100111111111111\ldots$, say, as $0.0101101000000000000\ldots$. Conversely, if two binary fractions differ by at least $1/2^b$ for an integer b then they must differ on one of their first b bits after the binary point. It follows that, since s_i differs from s_{i-1} by $p_{i-1} \geq p_i$ for $i > 1$ and from s_{i+1} by p_i for $i \leq n$, s_i is uniquely distinguished by its first $\lceil \lg(1/p_i) \rceil$ bits after the binary point. Taking the first $\lceil \lg(1/p_i) \rceil$ bits of s_i to the ith outcome, the expected codeword length is

$$\sum_{i=1}^{n} p_i \lceil \lg(1/p_i) \rceil < \sum_{i=1}^{n} p_i \lg(1/p_i) + 1 = H(P) + 1\,.$$

Codes in which every codeword is self-delimiting are called *prefix-free* (sometimes abbreviated to only "prefix"), because no codeword can be a prefix of another codeword, or *instantaneous*, because we can tell as soon as we've reached the end of a codeword. Prefix-free codes built with Shannon's construction are called, naturally enough, Shannon codes. Shortly after Shannon published his article, Robert Fano proposed a construction that recursively divides the p_is into two subsets whose sums are as close as possible.

Even before he was the father of information theory, Shannon had made important contributions to electrical engineering: for example, in his master's thesis at MIT, written when he was 21, he introduced the idea of modelling digital circuits with Boolean logic (and included a diagram of a 4-bit full adder); if you think Question 3 on Assignment 2 was hard, imagine what it would have been like without the notion of logic gates! Suffice to say, like Turing (with whom he met at Bell Labs), Shannon was a genius. Nevertheless, Shannon codes can be sub-optimal, meaning the expected codeword length is not always minimum. For example, for $P = \frac{6}{16}, \frac{5}{16}, \frac{5}{16}$, Shannon's construction produces a code with all codeword lengths equal to

$$\lceil \lg(16/6) \rceil = \lceil \lg(16/5) \rceil = 2\,,$$

but if we assign the first outcome codeword 0 and the third and fourth codewords 10 and 11, then the expected codeword length is

$$(6/16) + 2((5/16) + (5/16)) = 1.5\,.$$

People realized immediately that Shannon's article was a masterpiece and by 1951 Fano was teaching a graduate on information theory. He gave the students the option of doing a project or writing a final exam, and one of the students, David Huffman, decided to find an efficient algorithm for building optimal prefix-free codes. The story goes that he failed repeatedly and was about to give up when inspiration struck and he found the famous greedy algorithm now named after him, which he published in 1952 and we're going to cover in this lecture.

Before we try to understand Huffman's algorithm, however, I'd like us to warm up by developing a greedy algorithm for another problem. Suppose that during the last lecture I tried to demonstrate MergeSort by taking an unsorted sequence, splitting it half, and sending the two subsequences to two students with instructions to continue the recursion and send me their sorted subsequence when it was ready. Somehow wires got crossed, however, and some other students sent their sorted subsequences directly to me instead of giving them back to the people they'd got them from. To save time, I decide to merge the subsequences myself, using the code I showed you in the last lecture that merges two sorted subsequences of lengths n_1 and n_2 into a single sorted sequence of length $n = n_1 + n_2$ in $O(n)$ time. The subsequences are of many different sizes, however, and I want to know in which order I should merge them to minimize the time I spend.

Faced with this problem of planning how to perform pairwise merges of subsequences, with each merge taking time proportional to the combined length of the two merged subsequences, an obvious first step is to merge the two shortest subsequences. Since this minimizes the time taken by the first merge, in some sense it's the obvious *greedy* choice.

Suppose we start with n subsequences — ignoring the fact that in the last lecture n was the length of the original sequence and thus the total length of the subsequences — so after we merge the two shortest, we have $n - 1$ subsequences (including the result of that first merge). For now, let's concentrate on proving only that there exists *some* strategy — maybe greedy, maybe not — that merges those $n - 1$ subsequences such that the total time (including the first merge) is the minimum possible for any pairwise merging strategy.

Suppose, for example, that we've asked my friend Jérémy (an expert on sorting, as I mentioned in the last lecture) to merge our subsequences for us, but he's been delayed and we've decided to choose and perform the first merge ourselves. How can we be sure that after we merge the two shortest subsequences, when Jérémy does eventually arrive he'll say something like

"That's not exactly what I would have done, but I can work with it; we can still get an optimal solution (so the total time spent merging is minimized)."

and not something like

> "Sacre Bleu, what have you done? All is lost! There's no way to reach an optimal solution now!" ?

(Jérémy is French, as you may have guessed.)

Any strategy for merging the subsequences pairwise can be viewed as a strictly binary tree (that is, every internal vertex has exactly 2 children), with the input subsequences at the leaves, each internal being the merge of its children, and the root being the complete sequence. Symmetrically, any such tree can be viewed as a strategy for merging the subsequences. Notice that an element of the input subsequences for leave v is processed during the merges resulting in the subsequences for v's proper ancestors (that is, the ancestors of v excluding v itself), taking constant time during each of those merges. Therefore, the total contribution of all the elements in v's subsequence to the time for merging, is proportional to the length of that subsequence times v's depth. (Yes, this is similar to our analysis of QuickSort; that's deliberate.)

If we assign each leaf weight equal to the length of its subsequence, then a strategy according to which the time spent merging is minimized, corresponds to a tree in which the sum of the products of the leaves' weights and depths is minimized. Let's call that sum the tree's *cost*. Suppose we are given a tree T_{Opt} that corresponds to an optimal merging strategy, so it has minimum cost.

If a leaf with a smaller weight is higher in T_{Opt} than a leaf with a larger weight, then we can swap them and reduce T_{Opt}'s overall cost, contrary to the assumption its cost is minimum. Therefore, if we choose to merge subsequences with lengths w_i and w_j because those are the shortest, we can assume T_{Opt} has two leaves at its bottom level with weights equal to w_i and w_j. (It can't have a single leaf at its bottom level because it's strictly binary.)

Swapping leaves at the same level doesn't change the cost of the tree, and neither does swapping leaves with the same weight, so there exists a tree T'_{Opt} with the same cost as T_{Opt} in which the leaves for the subsequences we merge first are siblings (at the bottom level). In other words, there indeed exists some optimal merging strategy that starts by merging the two subsequences we merge — meaning Jérémy will be able to take over and reach an optimal solution.

Now consider the algorithm that merges subsequences into a single subsequence by repeatedly merging the shortest two subsequences (breaking ties arbitrarily). We have just shown that if we merge the shortest two of n subsequences and then merge all the resulting $n-1$ subsequences optimally, then we use the minimum possible time for merging. It follows that if our algorithm is optimal for merging $n-1$ subsequences, then it is optimal for merging n subsequences.

Clearly our algorithm is optimal for 0 or 1 subsequences (for which it has nothing to do) or even two subsequences (for which it makes the only possible merge). Therefore, by induction, it is optimal for any number n of subsequences.

Theorem 6 *Given a set of sorted subsequences to merge pairwise into a single sequence, with each merge taking time proportional to the combined length of the two merged subsequences, we can minimize the total time used by repeatedly merging the two shortest subsequences.*

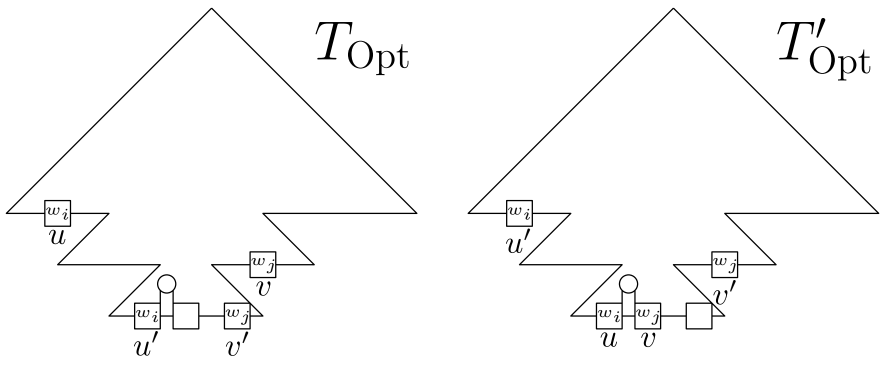


Figure 7.2: If we merge subsequences with lengths w_i and $w_j$corresponding to leaves u and v, then there must be leaves u' and v' at the bottom level of T_{Opt} with those weights. We can swap u and v with u' and v' to have u and v at the bottom level and swap them with other leaves at the bottom level to make them siblings in a new tree T'_{Opt} with the same cost as T_{Opt}.

Considering again pairwise merging strategies as corresponding to binary trees, with optimal strategies corresponding to trees with minimum cost (where a tree's cost is the sum of the products of its leaves' weights and depths), we can rephrase Theorem 6 as follows:

Theorem 7 *Given positive weights, the following algorithm builds a binary tree whose leaves are assigned those weights, in some order, such that the sum of the products of the leaves' weights and depths is minimized:*

1. *we create a set of vertices with those weights;*
2. *until there is only one vertex in the set, we repeatedly*
   (a) *remove the two vertices u and v with the smallest weights in the set,*
   (b) *make u and v the children of a new vertex t with weight equal to the sum of u's and v's weights,*
   (c) *insert t into the set;*
3. *we return the single vertex remaining in the set, as the root of the tree.*

An obvious way to implement our algorithm in Theorem 7 is to keep vertices with the current weights in a priority queue, in non-decreasing order by weight. When we extract the two vertices with the smallest weights w_i and w_j, we make them the children of a new vertex with weight w_i+w_j, which we insert into the priority queue. Using a min-heap as a priority queue, the algorithm then takes $O(n \log n)$ time.

Now that we have warmed up, we are ready to consider the problem of designing an optimal prefix-free code, as Huffman did. Specifically, given a probability distribution $p_1, \ldots, p_n$, we want to associate binary codewords to the p_is such that no codeword is a prefix of any other codeword and, when the ith codeword is chosen with probability p_i, the expected codeword length is minimum.

The key to this problem is to realize that a collection of n binary codewords in which no one codeword is a prefix of any other codeword, can be viewed as a binary tree with edges from parents

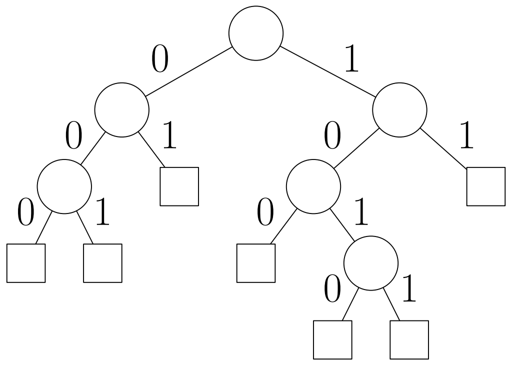


Figure 7.3: A binary tree corresponding to a prefix-free code with codewords 000, 001, 01, 100, 1010, 1011, 11.

to left children with 0s, edges from parents to right children with 1s, and the paths from the root to the leaves labelled with the codewords. Figure 7.3 illustrates this correspondence.

From this perspective, building an optimal prefix-free code is equivalent to building a binary tree whose leaves are assigned weights $p_1, \ldots, p_n$, in some order, such that the sum of the products of the leaves' weights and depths is minimized. Because we can assume all the probabilities are positive, in fact we have already solved this problem with Theorem 7! Our supposed warm-up was actually a derivation of Huffman's algorithm. As one of my former supervisors would say, "I was *tricking* you!" I hope this presentation is easier to understand than the traditional one: last year I had a student spontaneously suggest merging the two shortest lists as a first step, and I've never had anyone spontaneously suggest something like "make the two leaves labelled with the least likely characters in the alphabet the children of a new vertex assigned a probability equal to the sum of theirs". Figure 7.4 gives an illustration of Huffman's algorithm.

In order to fill some space and hide my trick from people glancing ahead in these notes (who might be suspicious if we finish too quickly after proving Theorem 7), now we will discuss Van Leeuwen's algorithm, which is a version of Huffman's algorithm that runs in linear time when the probabilities are given already sorted (or can be represented by integers in a range of size polynomial in n, in which case we can sort them in linear time ourselves). This is in contrast to using a priority queue, which uses $\Omega(n \log n)$ time with most common implementations of priority queues. Van Leeuwen's algorithm is also a nice example of how the clever use of data structures, even standard ones, can significantly speed up algorithms.

Suppose we are given weights $w_1, \ldots, w_n$ with $w_1 \leq \cdots \leq w_n$. We enqueue vertices with weights $w_1, \ldots, w_n$ in order in a queue Q_1, such that the vertex with weight w_1 is at the head of Q_1 and the vertex with weight w_n is at its tail. We create an empty queue Q_2 and repeat the following steps until there is only one vertex left in the queues:

1. dequeue a vertex with weight x from whichever of Q_1 and Q_2 has a lighter vertex at its head (or from whichever has a vertex at its head, if one is empty);
2. dequeue a vertex with weight y from whichever of Q_1 and Q_2 has a lighter vertex at its head (or from whichever has a vertex at its head, if one is empty);

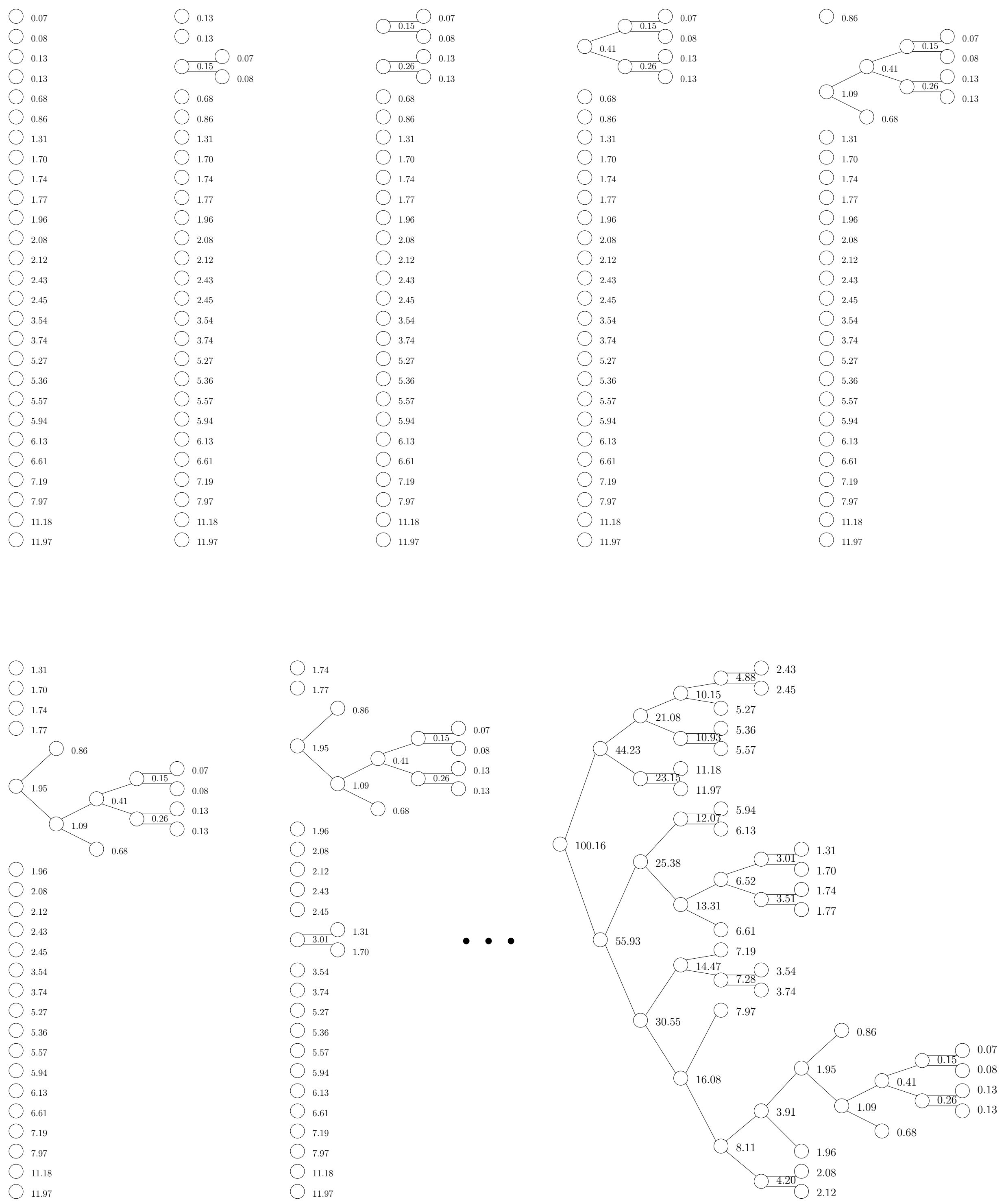


Figure 7.4: The first few steps of Huffman's algorithm, implemented with a priority queue, and the complete tree **(bottom right)**. The weights are the approximate frequencies (as percentages) of the characters A through Z and SPACE in English text. (They sum to 100.16 instead of 100 because of rounding.)

3. make those vertices the children of a new vertex with weight $x + y$;
4. enqueue that new vertex in Q_2.

Since enqueuing and dequeuing take constant time, the algorithm takes $O(n)$ time overall.

As long as both queues are always sorted, dequeuing from whichever has a lighter vertex at its head is equivalent to extracting from a priority queue. Obviously Q_1 stays sorted, because we only dequeue from it and never enqueue to it. To see why Q_2 stays sorted, assume for the sake of a contradiction that at some point we enqueue a vertex v to Q_2 with a weight $x + y$ that is smaller than weight of the vertex u ahead of it in Q_2, and that previously both queues were always sorted.

Since Q_2 is initially empty and we enqueue only vertices with weights that are the sums of the vertices they are replacing, u's weight is $x' + y'$ for some weights x' and y' of vertices that were dequeued before the vertices with weights x and y that v is replacing. If both queues were always sorted up to this point, however, and we dequeued vertices with weights x' and y' before vertices with weights x and y, however, then $x' + y' \leq x + y$, so v is heavier than u.

Theorem 8 *If we are given a distribution with n sorted probabilities, then we can build an optimal prefix-free code for them in $O(n)$ time.*

As promised, and to fill some more space, now we will see the proof (more or less) of the lower bound in the Noise Coding Theorem: given $P = p_1, \ldots, p_n$, it is *not* possible to assign self-delimiting binary codewords to the n possible outcomes such that the expected codeword length is less than $H(P)$.

Let's assume the p_is are positive rationals, so each p_i can be expressed as a_i/b_i for some positive integers a_i and b_i. Let m be a sufficiently large common multiple of the b_is and let $m_i = a_i m/b_i$, so $m_1 + \cdots + m_n = m$. For the sake of a contradiction, assume it is possible to assign self-delimiting binary codewords $w_1, \ldots, w_n$ to the outcomes such that the expected codeword length

$$\sum_i p_i |w_i| < H(P) - \epsilon$$

for some positive constant ϵ.

Consider an alphabet $\{c_1, \ldots, c_n\}$ and all the possible strings of length m in which c_i occurs m_i times, for each i. There are $\binom{m}{m_1,\ldots,m_n}$ such strings, where the multinomial coefficient

$$\begin{aligned}
&\binom{m}{m_1, \ldots, m_n} \\
&= \binom{m}{m_1}\binom{m - m_1}{m_2} \cdots \binom{m - m_1 - \cdots - m_{n-2}}{m_{n-1}}\binom{m - m_1 - \cdots - m_{n-2} - m_{n-1}}{m_n} \\
&= \left(\frac{m!}{(m - m_1)!m_1!}\right)\left(\frac{(m - m_1)!}{(m - m_1 - m_2)!m_2!}\right) \cdots \left(\frac{(m - m_1 - \cdots - m_{n-2} - m_{n-1})!}{(m - m_1 - \cdots - m_{n-2} - m_{n-1} - m_n)!m_n!}\right) \\
&= \frac{m!}{m_1!m_2! \cdots m_n!}\,.
\end{aligned}$$

Notice this is equal to $2^{\lg m! - \lg m_1! - \cdots - \lg m_n!}$.

By Stirling's formula,

$$\begin{aligned}
&\lg m! - \lg m_1! - \cdots - \lg m_n! \\
&= (m \lg m - m \lg e + O(\log m)) - \sum_i (m_i \lg m_i - m_i \lg e + O(\lg m_i)) \\
&= m \lg m - \sum_i m_i \lg m_i - m \lg e + \sum_i m_i \lg e + O(\log m) - \sum_i O(\log m_i) \\
&= \sum_i m_i \lg m - \sum_i m_i \lg m_i - \sum_i m_i \lg e + \sum_i m_i \lg e + O(\log m) - \sum_i O(\log m_i) \\
&= \sum_i m_i \lg(m/m_i) + O(\log m) - \sum_i O(\log m_i) \\
&= m \sum_i (m_i/m) \lg(m/m_i) + O(\log m) - \sum_i O(\log m_i) \\
&= mH(P) + O(\log m) - \sum_i O(\log m_i) \\
&\geq mH(P) - O(n \log(n/m))\,.
\end{aligned}$$

For sufficiently large m, this is greater than $m(H(P) - \epsilon/2)$, so we can assume we are considering more than $2^{m(H(P)-\epsilon/2)}$ possible strings.

Suppose we assign each c_i codeword w_i and encode each of those possible strings by concatenating the codewords of its characters. Then we encode each such string with a distinct binary string consisting of

$$\sum_i m_i |w_i| = m \sum_i p_i |w_i| < m(H(P) - \epsilon)$$

bits — but there are fewer than $2^{m(H(P)-\epsilon)+1} - 1$ of those and, for sufficiently large m,

$$2^{m(H(P)-\epsilon/2)} > 2^{m(H(P)-\epsilon)+1} - 1\,,$$

so we have a contradiction.

For a course on algorithms, you don't really need to know why the Noiseless Coding Theorem holds, but the proofs of the upper and lower bounds aren't really all that hard (and used some things we've seen before) and the result is generally considered deep and beautiful, so I thought it worth including them. Understanding the full proof of the Noiseless Coding Theorem may also help you understand an apparent contradiction: that theorem says we can't beat the entropy and that Huffman coding is optimal, but you'll often hear people saying arithmetic coding beats Huffman coding and that more sophisticated schemes do even better.

To see that this isn't really a contradiction, it's important to remember that Shannon and Huffman were assuming we know the probability distribution over the universe of possible entire messages. That's usually not the case and, when people actually use Huffman coding, they're usually taking a string, counting how many times each character in the alphabet occurs in that string, assigning each character probability proportional to its frequency, building a Huffman code for the resulting probability distribution, and then using that code to encode the string character by character. That's quite different from trying to encode the whole string at once!

As we just saw, if the string is long enough and the characters are shuffled randomly, then we can't achieve an average codeword length of $H(P) - \epsilon$ for the characters, where P is the distribution of characters in the string. If the characters are shuffled or if they're sampled from a multiset with replacement — for example, drawn from a hat containing slips of paper with the characters written one them, possibly with a different number of slips for each character, and then returned to the hat — then Huffman coding is (essentially) optimal among all codes that encode the string character by character, using an integer number of bits for each character. Such codes are sometimes called character codes or, when prefix-free, instantaneous codes (since we can decode each character when we reach the end of its codeword), and they can be beaten by schemes, such as arithmetic coding, that can encode several characters together.

More sophisticated compression schemes also take advantage of the fact that the choice of each character is not independent and identically distributed (often abbreviated to "iid"). For example, a "q" in in English is usually followed by a "u", and if we've just seen "th" then the next character is probably an "e" or an "a", or maybe a space or an "o" or an "r", but almost certainly not a "t", even though "t" is a fairly common in English generally. Those schemes can beat Huffman coding when they use a contextual model and it's used as a memoryless character code.

Arithmetic coding can also be used with a contextual model; in fact, it was invented partly to determine which is the best model, according to the principle of minimum description length: of a class of models for a string, the best one is the one that minimizes the total size of the model and of the encoding of the string with respect to the model. You may hear more about this in courses on machine learning.

Chapter 8

Minimum Spanning Trees

Building a minimum spanning tree of a connected, edge-weighted graph is a classic problem, and a classic example of a problem we can solve with greedy algorithms. There are many greedy algorithms for building minimum spanning trees, and we'll look at three: Kruskal's, Prim's and Borůvka's. They have different advantages and disadvantages which you should consider when choosing one for a particular situation.

Recall that a graph is connected if and only if there exists a path from any vertex to any other vertex. (For this lecture, we'll consider only undirected graphs.) An edge-weighted graph has a cost associated with each edge; without loss of generality, we can assume all the weights are positive. (If some of the edges have negative weights or weight 0, then we can include all those in our solution without increasing its total cost, and then consider as single vertices the connected components of the resulting graph.) In a connected graph with positive edge weights, a connected subgraph with minimum weight that spans all the vertices — that is, includes them all — will necessarily be a tree. (If it included a cycle, we could remove one of the edges in that cycle and reduce the subgraph's weight without disconnecting it.) Such a subgraph is thus called a *minimum spanning tree* (MST).

There are many applications for which MSTs are useful, which is one reason there are so many algorithms, and why they have been rediscovered so often: Borůvka developed his algorithm in 1926 for designing an electrical grid for part of the eastern Czech Republic (then part of Czechoslovakia) and, according to Wikipedia (`https://en.wikipedia.org/wiki/Boruvkas_algorithm`) it was later rediscover by Choquet in 1938, by Florek et al. in 1951, and by Georges Sollin in 1965 (and is sometimes known as "Sollin's algorithm"); Prim's algorithm also first developed in Czechslovakia, by Jarnik in 1930; it was then rediscovered by Prim in 1957 and again by Dijkstra in 1959 (so it is sometimes known as "Jarnik's algorithm", or "Prim-Jarnik", or "Prim-Dijkstra" or "DJP").

For example, suppose you're the county planner in charge of paving some roads in your county such that it's possible to drive between any two towns along paved roads, and every road starts at a town and ends at another town without forking or intersecting other roads along the way.[*] Let's assume it is currently possible to drive between any two towns along dirt roads without leaving the county, you know the cost for paving each road and, although you need not worry about how long it will take to drive between towns along paved roads, you want to spend as little money as possible on paving. How do you choose which roads to pave?

We can model this as a graph, with the towns as vertices, the roads as edges, the cost of paving a road as the weight of the corresponding edge, and an optimal plan for paving as an MST of the graph. Let's start by considering what at first seems like the simplest algorithm for this, Kruskal's: first, we sort the edges by weight, breaking ties arbitrarily; then, we process the edges in non-decreasing order by weight, adding each edge to our current subgraph if it does not create a cycle and discarding it if it does.

To see why Kruskal's algorithm works, let's consider the standard form of a proof of correctness for a greedy algorithm:[†]

> **Before we take any steps, our (empty) subsolution can be extended to an optimal solution. Assume that, after $i \geq 0$ steps, our subsolution can be extended to an optimal solution S. Then we show that after $i + 1$ steps, our subsolution can be extended to an optimal solution S'. Therefore, by induction, we obtain an optimal solution.**

To show that our subsolution after $i+1$ steps can be extended to an optimal solution, consider the edge e we process in our $(i + 1)$st step.

If we discard e then, because we discard edges only when they create a cycle with our current subsolution and S extends our current subsolution — so it contains all of the edges we have taken so far and none of the edges we have discarded — S cannot contain e, so S itself extends our subsolution after $i+1$ steps (meaning $S' = S$). If we take e and S includes e then, again, S extends our subsolution after $i + 1$ steps (and $S' = S$). Finally, if we take e and S does not include e, then we must show how to change S to obtain a solution S' that does include e but does not cost more. We will do this in the normal way, by an *exchange argument*: proving we can exchange e for another edge in S of equal or greater weight without disconnecting the subgraph in S.

If S does not already include e, then adding e to S creates a cycle. There must be some edge f in this cycle we have not already considered (because S agrees with our subsolution after i steps). Since we consider the edges in non-decreasing order by weight, f's weight must be at least as great as e's. Therefore, by replacing f with e, we obtain a solution S' that still spans the graph and has total weight no more than S.

To use Kruskal's algorithm, it's important that you know the whole graph for which you're trying to build an MST. It's reasonable to assume a county planner has a map of their county, of

[*]If we consider intersections and allow plans in which pavement on roads can stop at intersections instead of at towns, then we have an instance of the minimum Steiner-tree problem, which is another of those "BOOM" problems essentially equivalent to finding Hamiltonian paths.

[†]This is a more formal version of the argument in the last lecture that my friend Jérémy can show up after we take a step and still reach an optimal solution.

course, but suppose you're trying to choose a subset of communication links between computers in a network such that a message can be sent from any computer to any other through those links, the links have various costs, and you find out how a computer is linked to the rest of the network only when you have selected links allowing it to exchange messages with your computer.

Prim's algorithm can deal with this version of the MST problem because, while Kruskal's algorithm builds subsolutions that are forests, Prim's subsolutions are always trees: first, we choose a vertex from which to start our MST; then, at each step, we consider the edge with lowest weight incident to our current tree which we have not already considered and add it to our tree unless it creates a cycle, in which case we discard that edge.

To see why Prim's algorithm works, we again follow the standard form:

> **Before we take any steps, our (empty) subsolution can be extended to an optimal solution. Assume that, after $i \geq 0$ steps, our subsolution can be extended to an optimal solution S. Then we show that after $i+1$ steps, our subsolution can be extended to an optimal solution S'. Therefore, by induction, we obtain an optimal solution.**

(I'm repeating this because I want you to be able to remember it during an exam.) Again, we will use an exchange argument to show that, if we take an edge e in the $(i+1)$st step and S does not include e, then we can change S into a solution that includes e without increasing the total weight.

The argument again starts with adding e to S and considering the resulting cycle, but this time we cannot claim that any edge in that cycle we have not yet considered must have weight at least as great as e's, because some of them may not be incident to our tree after i steps. Nevertheless, since the cycle includes e, and one of e's endpoints is in our tree after i steps but the other is not (otherwise our taking e would create a cycle in that tree), then *some* edge f in the cycle is not in our tree after i steps but is incident to it. Therefore, since we consider the edges incident to our tree in non-decreasing order by weight, f's weight must be at least as great as e's, so by replacing f by e we obtain an optimal solution S' that extends our subsolution after $i+1$ steps.

To get a broader perspective, I encourage you to read about Kruskal's and Prim's algorithms in *Introduction to Algorithms.* For consistency, since I used it in the video, I've also copied their example of how those algorithms produce MSTs (which I hope falls under "fair use" of copyrighted material), in Figures 8.1 and 8.2.

Finally, suppose that all the computers in the network want to work together to build an MST quickly, but know only about the links incident to computers with which they they can exchange messages. In this case, we can use Borůvka's algorithm (which I don't think *Introduction to Algorithms* covers): like Kruskal's algorithm, a subsolution is a forest and not necessarily a tree; like Prim's algorithm, in each step, the computers in each tree consider all the edges incident to that tree and add the one with lowest weight, unless it creates a cycle.

For the sake of simplicity, let's assume either the links all have slightly different costs, or there's some locking mechanism to prevent the computers in two trees simultaneously adding edges with equal weight that create a cycle. If the computers in each tree take 1 time unit to choose the edge

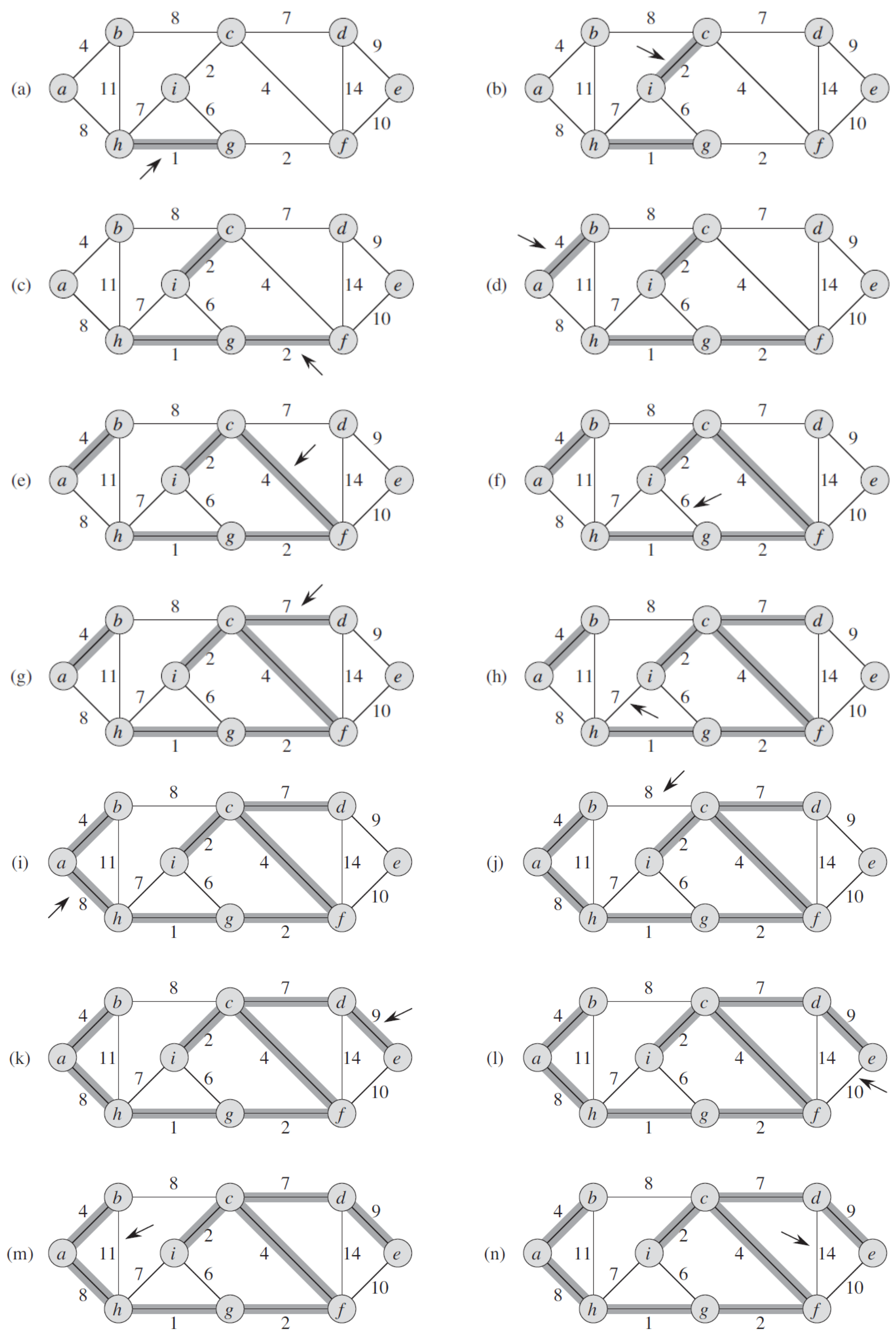


Figure 8.1: The example of Kruskal's algorithm from *Introduction to Algorithms*: "The execution of Kruskal's algorithm on the graph from Figure 23.1. Shaded edges belong to the forest. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees."

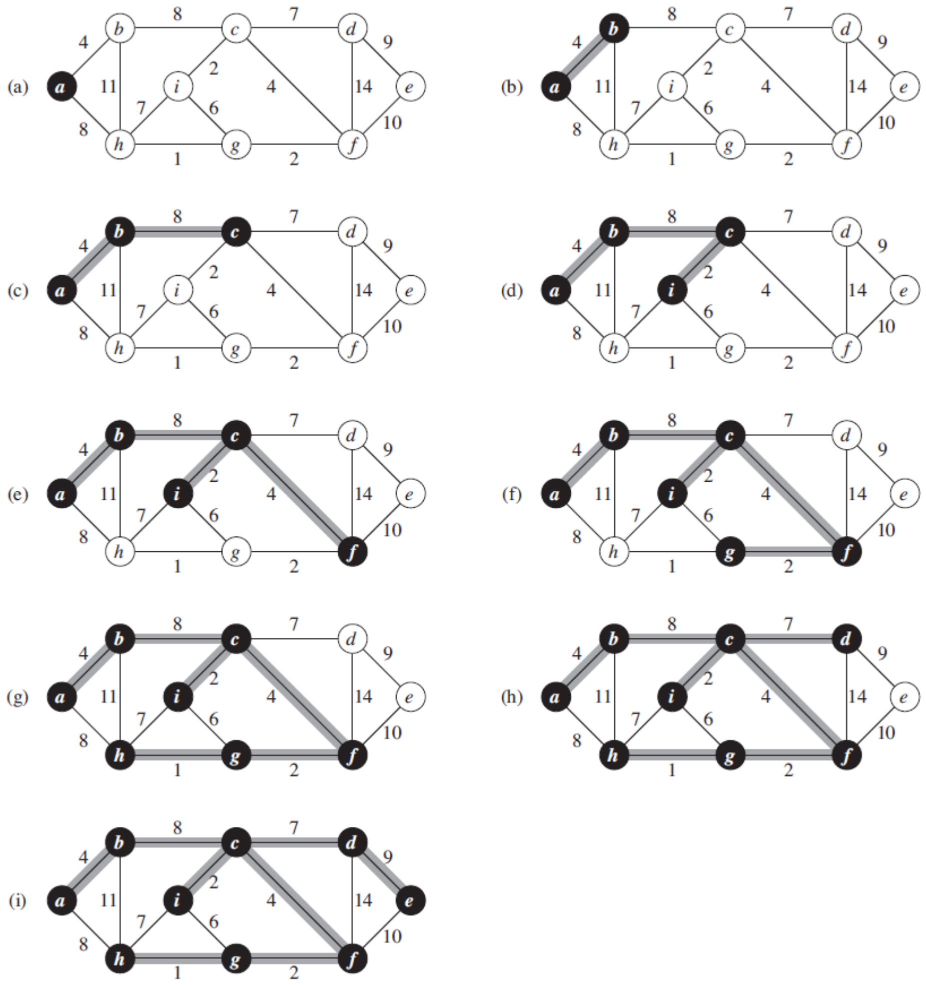


Figure 8.2: The example of Prim's algorithm from *Introduction to Algorithms*: "The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is a. Shaded edges are in the tree being grown, and black vertices are in the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph [Travis: don't worry what this means, we'll learn about cuts if we have time to cover network flows], and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree, since both are light edges crossing the cut."

incident to that tree with lowest weight and add it to the tree, then after 0 time units the trees consist of 1 vertex each; after 1 time unit the trees consists of at least 2 vertices each; in general, after i time units the trees consist of at least 2^i vertices each; and we have an MST in $O(\log n)$ time units, where n is the number of vertices in the graph. That is, although building an MST takes $\Omega(n)$ work, it can be done in $o(n)$ time if we can use parallelism.

Although the strength of Borůvka's algorithm is that computers in the different trees can work in parallel, it's easiest to see why it is correct by considering the computers in one tree adding one edge in isolation. In this case, we can consider the other trees as single vertices, so the correctness of a step in Borůvka's algorithm follows from the correctness of Prim's algorithm (or of Kruskal's algorithm, from a different perspective).

Let's now look at how to implement each algorithm. With Prim's algorithm we need a priority queue of the edges incident to our current tree. If we use a min-heap with $\Theta(\log m)$ time to extract the min or insert a new element, where m is the number of edges, then Prim's algorithm takes $\Theta(m \log m) = O(m \log n)$ time. Detecting when adding an edge would create a cycle is relatively easy: we keep all the edges in our tree coloured a certain colour and, whenever we add an edge to the tree, we colour its endpoint that was just added to the tree and consider all the edges incident to that vertex; we discard any that have both endpoints coloured and insert into the priority queue any that have only one endpoint coloured.

We can also implement Borůvka's algorithm in $O(m \log n)$ time: we initially colour all the vertices different colours and then proceeding in $O(\log n)$ rounds, maintaining the invariant that all the vertices in any one tree are all the same colour, and a different colour than the vertices in any other tree; in each round, we scan all the edges in the graph; if an edge e's endpoints are two different colours and one of the endpoints is red, for example, and e's weight is the smallest we've seen so far during this round for any edge with a red endpoint, then we keep e as our candidate edge to add to the red tree (discarding any previous candidate); when we finish the scan, we have the edge with lowest weight incident to each tree; we then add those edges — taking some care not to create a cycle by adding two edges with the same weight incident to the same tree (two candidates with endpoints that share a colour) — and recolour vertices so all the vertices in each tree are the same colour (perhaps with breadth-first or depth-first traversals of the trees). This takes $O(m + n) = O(m)$ time per round, or $O(m \log n)$ time in total.

The most interesting implementation is of Kruskal's algorithm. At first it seems the simplest, because we need only sort the edges once. It is not so obvious how to detect when adding an edge would create a cycle, however: we cannot use a single colour, as with Prim's algorithm; but if we use many colours, like Borůvka's algorithm, then we must be careful not to spend too much time recolouring.

Suppose we initially colour all the vertices different colours and maintain the invariant that all the vertices in any one tree are all the same colour, and a different colour than the vertices in any other tree, and we know the count of how many vertices are in each tree. Whenever we consider an edge, if its endpoints are the same colour then we discard it; otherwise, we choose the smaller of the two trees it connects and recolour all its vertices to match the vertices in the larger tree. With a fairly simple implementation this takes time linear in the size of the smaller tree, and it is not hard to see that any vertex is recoloured $O(\log n)$ times. It follows that, after sorting the edges, we spend $O(n \log n)$ time building an MST with Kruskal's algorithm. Since we can sort the edges in $O(m \log m) = O(m \log n)$ time, we get an $O(m \log n)$ bound for Kruskal's algorithm as well.

As you'll remember from the lecture on sorting, however, in some models or in some circumstances we can sort the m edges in $o(m \log n)$ time. Therefore, it makes sense to try to optimize the post-sorting construction and, for that, we'll now briefly cover the *union-find* data structure. This data structure maintains a collection of disjoint sets with a representative for each set, and supports the following operations:

`create`(x) creates a new set $\{x\}$ (when x is not already in any set);
`union`(x, y) merges the sets containing x and y (unless they are already in the same set);
`find`(x) returns the representative of the set containing x.

It shouldn't be too hard to see how to use this data structure to implement Kruskal's algorithm: whenever we consider an edge, we `find` the representatives of its endpoints and, if they are the same, we discard the edge; if not, we `union` the sets containing the endpoints. It also shouldn't be too hard to see how, as long as our sets contain a total of n elements, we can support all of the operations in $O(\log n)$ time, using the idea for implementing Kruskal's algorithm in $O(m \log n)$ time: we keep a coloured tree for each set, whose vertices are the elements in that set, and a representative vertex associated with each colour; when we `union` two elements, we add an edge between their vertices and recolour the smaller of their two trees.

The standard implementation of union-find actually works much faster than this. We still keep the elements of a set in a tree, but now the tree is rooted and each vertex points to its parent (with the root pointing to itself). We consider the root to be the representative of the set and, when we `union` two sets, we switch the pointer at the root of the smaller tree from pointing to itself, to pointing to the root of the larger tree. The main change is in how we implement `find`, using an idea called *path compression*: when we perform `find`(x), we follow the pointers up the path from x to the root of its tree and, as we go, we push x's ancestors on a stack so that, when we finally reach the root of the tree, we can reset all of their pointers to the root. This increases the worst-case time of a `find` by a constant factor, but it greatly decreases the *amortized* time.

I'm not sure if you've seen amortized analysis in previous courses. The simplest example is an expandable array, which is useful when we are receiving elements and want to store them in an array but don't know how many there are or how much space to allocate. By starting with a small array and doubling the size whenever it fills up, we never use more than a constant times as much space as we need and, although the worst-case time to receive an element is linear in the number of elements we have received — to create the new array, copy all the elements into it, and free the old array — the total time is linear in the number of elements and so the time to receive and process each element is constant when amortized over all of the elements. To see why, suppose we charge 3 units to receive each element, spend 1 to insert it into the current array, and save 2; then, when an array of n elements is full, we have $2(n/2) = n$ units saved to pay for copying each element into a new array.

The analysis of union-find with path compression is quite detailed, and even *Introduction to Algorithm* presents only a simplified version, showing that it makes Kruskal's algorithm take $O(m \log^* n)$ time after sorting the edges, where $\lg^* n$ is the number of times we need to apply lg to

n to get down to 1. The function $\lg^* n$ grows very slowly; for example, even though $2^{2^{2^{2^2}}} \approx 2{\cdot}10^{19728}$,

$$\lg^* 2^{2^{2^{2^2}}} = \lg^* 2^{2^{2^2}} + 1 = \lg^* 2^{2^2} + 2 = \lg^* 2^2 + 3 = \lg^* 2 + 4 = 5.$$

In fact, m operations on an initially empty collection of sets, of which n operations are `create`, takes $O(\alpha(m,n))$ time, where α is the Inverse Ackermann Function, which grows even more slowly than $\lg^*$.

Chapter 9

More Greedy Algorithms

We started our study of greedy algorithms with Huffman coding so I could trick you into rediscovering Huffman's algorithm while pretending we were warming up with a simple exercise about merging sorted lists. Most other courses on algorithms start with simpler problems, not known to have stumped Shannon ("Father of the Information Age"). Now that we've seen Huffman's, Kruskal's, Prim's and Borůvka's algorithms, let's look at a couple of those easy problems — and some harder versions of them, to keep things interesting.

The first problem *Introduction to Algorithms* shows how to solve with a greedy algorithm is Activity Selection: we are given a list of activities with their start times and finish times; we can think of an activity with start time s and finish time f as a half-open interval $[s, f)$; an activity $[s, f)$ is compatible with another activity $[s', f')$ if and only if their intervals do not intersect, $[s, f) \cap [s', f') = \emptyset$; we want to choose the largest possible subset of pairwise-compatible activities.

The example of Activity Selection I have in my edition of *Introduction to Algorithms* has these activities: $[3, 5), [5, 7), [12, 14), [1, 4), [8, 11), [5, 9), [6, 10), [8, 12), [3, 8), [0, 6), [2, 13)$. For example, if your school-day lasts from 9:30 am to 4:30 pm, 0 means 9:30 and 14 means 4:30, then $[5, 7)$ could be lunch from 12:00 to 1:00 and the other activities could be lectures, seminars, tutorials, sports, naps, etc.

Assuming you can do only one thing at a time but you can get from one activity to another instantly — hooray for online learning! — and you value all activities equally — even 3110 lectures, and even lunch! — how can we choose your activities so that you do as many as possible?

Remember that greedy algorithms usually start by sorting their inputs in some way. I listed the activities in order by length, with ties broken by start time, but that's not always the best way to sort them. For example, if your activities are $[0, 3), [2, 4), [3, 6)$ then it's better to take $[0, 3)$ and $[3, 6)$ than only $[2, 4)$, even though that's the shortest. How else can you sort the activities? The obvious other ways are by start time — which isn't optimal, in cases such as $[0, 14), [1, 2), [2, 3), \ldots, [12, 13)$ — and by finish time.

If you try a few examples, it seems that processing the activities in order by finish time, from earliest to latest, and scheduling each activity if it is compatible with all the activities you've already scheduled and ignoring it otherwise, always produces an optimal schedule. (Ok, this is symmetric processing the activities by start time, from latest to earliest — just think of time as running backwards.) Why is this the case?

> **Before we take any steps, our (empty) subsolution can be extended to an optimal solution. Assume that, after $i \geq 0$ steps, our subsolution can be extended to an optimal solution S. Then we show that after $i + 1$ steps, our subsolution can be extended to an optimal solution S'. Therefore, by induction, we obtain an optimal solution.**

Suppose that in the $(i + 1)$st step we consider activity $[x, y)$. If we discard it then it must be incompatible with one of the activities we have already scheduled, in which case S cannot include it, so $S' = S$. If we schedule it and S includes it then, again, $S' = S$. The only interesting case is when we schedule $[x, y)$ and S discards it. If S does not schedule any activities incompatible with S, then we can add $[x, y)$ to S and achieve an improved solution (contrary to the assumption that S is optimal). Therefore, assume S schedules such an activity, $[x', y')$.

Notice that $x' < y$ (otherwise the two activities are not incompatible) and $y' \geq y$ (otherwise we would have considered $[x', y')$ before $[x, y)$), so S contains only one activity incompatible with $[x, y)$. If we remove $[x', y')$ from S, then the rest of S's activities are compatible with $[x, y)$, so we can add $[x, y)$ and achieve a solution S' containing $[x, y)$ that has as many activities as S and, so, is optimal.

Activity Selection is perhaps the simplest problem in a field known as *job scheduling*, which includes many "BOOM" problems. Imagine each activity can be started after a certain point (its *release time*), should be finished before a certain point (its *deadline*), has a certain profit (which can vary between activities), may depend on other activities having already been completed, and takes a certain amount of time (its *duration*) less than or equal to the difference between its deadline and its release time, with the duration depending on who among several people performs that activity — and you'll start to see why we're not going deeper into this topic in this course.

For Assignment 3, you showed how to sort quickly using a procedure that "given a sequence of n integers that can be partitioned into d non-decreasing subsequences but not fewer, does so in $O(n \log d)$ time". There actually exists such an algorithm, called Supowit's algorithm, that does this partitioning online and greedily. By "online" I mean that it processes the integers in order and has always partitioned those it has seen so far into a minimum number of non-decreasing subsequences.

For example, if the original sequence is $7, 2, 4, 3, 1, 2, 6, 9, 5, 8, 7, 4, 3$, how can you partition it online? The first 7 must start a subsequence, and the first 2 is less than 7 so it must start its own subsequence, but the first 4 could either start its own subsequence or follow the first 2; if the first 4 follows the first 2 then the first 3 must start its own subsequence, otherwise it can follow the first 2; etc.

For Supowit's algorithm, we keep the last element in each subsequence in a dynamic predecessor data structure (such as an AVL tree) with operations taking time logarithmic in the number of elements stored. We append each element x to the subsequence currently ending with x's predecessor among the last elements in the subsequences (that is, the largest element appearing last in a subsequence, that is at most x); if x is smaller than all of the current last elements, we start a new subsequence with it. If we append x to a subsequence, then we delete from the predecessor data structure the element that was the last element in the subsequence; in both cases, we insert x into the predecessor data structure. To process the entire sequence, we use $O(n \log d)$ time.

In our example, we start a subsequence with 7 and insert it into the predecessor data structure; search for the predecessor of 2 but find nothing, and start a new subsequence with 2 and insert it into the predecessor data structure; search for the predecessor of 4 and find 2, append 4 to the subsequence 2, delete 2 from the predecessor data structure and insert 4; search for the predecessor of 3 but find nothing, and start a new subsequence with 3 and insert it into the predecessor data structure; search for the predecessor of 1 but find nothing, and insert it into the predecessor data structure; search for the predecessor of 2 and find 1, append 2 to the subsequence 1, delete 1 from the predecessor data structure and insert 2; etc. Eventually, we obtain the following 5 subsequences:

$$\begin{array}{llll}
7, & 9 & & \\
2, & 4, & 6, & 8 \\
3, & 5, & 7 & \\
1, & 2, & 4 & \\
3\,. & & &
\end{array}$$

Why can we be sure it is impossible to partition our example sequence into fewer than 5 non-decreasing subsequences?

Before we take any steps, our (empty) subsolution can be extended to an optimal solution. Assume that, after $i \geq 0$ steps, our subsolution can be extended to an optimal solution S. Then we show that after $i + 1$ steps, our subsolution can be extended to an optimal solution S'. Therefore, by induction, we obtain an optimal solution.

Suppose that an optimal solution S extends our subsolution after i steps, when we have processed i integers. If S has the $(i + 1)$st integer x in the same subsequence that we put it in, the S extends our subsolution after $i + 1$ steps, so $S' = S$. If we start a new subsequence with x, then it cannot be appended to any of the existing subsequences, so S must start a new subsequence with x as well and, again, $S' = S$. The only case left to consider is when we append x to a subsequence A and S appends it to another (possibly empty) subsequence B.

Let a be the last element of A before x in our subsolution and let b be the last element of B before x in S. Let C and D be the suffixes in S of the subsequences that start A and B, x, respectively — meaning S includes subsequences A, C and B, x, D. Since we append x to the list ending with x's predecessor among the last elements of the subsequences when we consider x, we have $b \leq a \leq x$. Therefore, the first integer in C is larger than b, so we can swap C and x, D to obtain a solution with the same number of subsequences as S and containing the subsequences A, x, D and B, C. This solution extends our subsolution after $i + 1$ steps.

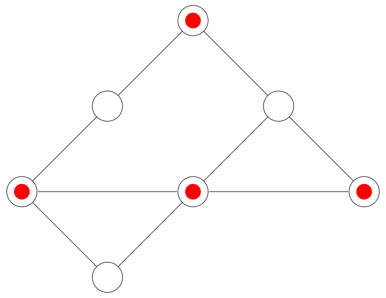


Figure 9.1: A vertex cover of size 4 (shown with red markers) of a graph with 7 vertices.

Interestingly, the problem of partitioning a sequence of integers into the minimum number of monotone subsequences — that is, with each subsequence either increasing or decreasing — is a "BOOM" problem. For various reasons, people who work in data compression and data structures would like to be able to partition sequences of integers into non-decreasing subsequences so as to minimize the entropy of the distribution of elements to subsequences. This isn't the same as minimizing the number of non-decreasing subsequences: for example, another solution to our example instance

$$\begin{array}{llll} 7, & 9 & & \\ 2, & 4, & 6, & 8 \\ 3, & 4 & & \\ 1, & 2, & 5, & 7 \\ 3\,. & & & \end{array}$$

has the same number of subsequences but the entropy of its distribution

$$(2/13)\lg(13/2) + (4/13)\lg(13/4) + (2/13)\lg(13/2) + (4/13)\lg(13/4) + (1/13)\lg(13/1) \approx 2.16$$

is slightly lower than the entropy of the distribution of the solution given by Supowit's algorithm,

$$(2/13)\lg(13/2) + (4/13)\lg(13/4) + (3/13)\lg(13/3) + (3/13)\lg(13/3) + (1/13)\lg(13/1) \approx 2.20\,.$$

As far as I know, no one knows whether partitioning sequences of integers into non-decreasing subsequences so as to minimize the entropy, goes "clink" or goes "BOOM". If you figure it out, either way, I promise to give you as many bonus marks as the faculty will let me.

Vertex Cover is a classic "BOOM" problem: given a graph on n vertices and an integer $k \leq n$, decide if there is a subset of k vertices such that every edge is incident to ("covered by") at least one vertex in that subset. (There's an analogous problem called Edge Cover for which we must select a subset of edges so as cover all the vertices. The way to keep them straight is to think "We're looking for a vertex cover (of the edges)".) Figure 9.1 shows a vertex cover of size 4 of a graph with 7 vertices.

Notice that, if you find a vertex cover consisting of fewer than k vertices, you can always add more vertices to get a vertex cover consisting of exactly k vertices, so we can write "of k vertices" instead of "at most k vertices". This is called a *decision problem* because it asks for a yes-or-no answer; the corresponding *optimization problem* is to find the smallest vertex cover. Theorists

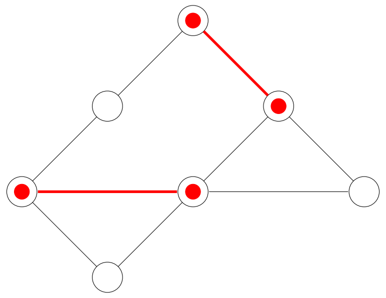


Figure 9.2: A maximal (but not maximum!) matching, and the vertex cover we get from including all the endpoints of its edges.

like thinking about decision problems because you can call the set of all strings encoding “yes” instances of Vertex Cover as a language, for example, and start using terms and ideas from formal language theory.

Even though there’s no known polynomial-time algorithm for finding the smallest vertex cover in a graph, there is an easy way to find a vertex cover that’s at most twice as big as the smallest: as often as we can, we choose an edge arbitrarily, then remove all the other edges incident to its endpoints; when we’re done, we have a *maximal matching*. (A matching is a subset of the edges such that no two share an endpoint. We say “maximal” instead of “maximum” because we can’t add any more edges, but that doesn’t necessarily mean there isn’t a larger matching.) If we then add *both* endpoints of each edge in our matching to our vertex cover, every edge is covered (because if an edge e were uncovered, then we could have added it to our matching) and our vertex cover is at most twice the size of the smallest one (because any vertex cover must include at least one endpoint of each edge in our matching, and none of those edges share endpoints).

Figure 9.2 shows a matching consisting of 2 edges for the graph shown in Figure 9.1, and the vertex cover — again of size 4 — that we get by choosing both endpoints of each edge in our matching. Although in this case we obtain a vertex cover as small as possible (I think), if we’d considered the edges in a different order when building our matching then we could have ended up with 3 edges instead of 2 (so our current matching is *maximal* but not *maximum*), and then our vertex cover would have 6 vertices instead of only 4.

When we’re faced with instances of “BOOM” problems in real life, we usually can’t just give up, so people have developed approaches to dealing with them. We can use approximation algorithms, that aren’t guaranteed to find optimal solutions but are guaranteed to find good ones (incidentally, there are better approximation algorithms for Vertex Cover, we just won’t see them in this course); we can use heuristics and hope for the best, because sometimes worst-case analysis is unduly pessimistic; we can try to solve them using brute force and big computers; or we can try to find characteristics of the instances we’re interested in that make them special and easier to solve. For example, although Vertex Cover goes “BOOM” in general, on trees it goes “clink” — in fact, there’s a greedy algorithm! (If you take Norbert’s 4th-year course on algorithms, you’ll see a *lot* more algorithms that work well on special cases.)

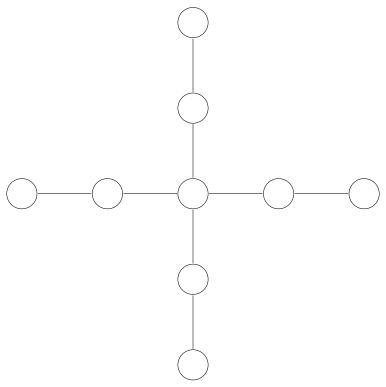


Figure 9.3: A tree with a vertex cover of size 4, but no vertex cover of size 4 including the central vertex.

The obvious greedy strategy for choosing a small vertex cover of a tree — start by adding the vertex with highest degree — doesn't always work optimally. For example, consider the cross-shaped tree shown in Figure 9.3: if we choose the center vertex because it has degree 4, then we still need to choose 4 more vertices to cover the edges incident to the leaves; if we choose the 4 vertices with degree 2, however, then we cover all the edges.

Instead, suppose we choose a leaf of the tree, add to our (initially empty) vertex cover the *other* endpoint v of the single edge incident to that leaf, delete that edge and all the other edges incident to v, and recurse on each tree of the resulting forest (ignoring all the vertices which are now isolated). Figure 9.4 shows an example of how this algorithm can work on a tree, with the numbers showing which vertices we consider as leaves (which need not be leaves when we start) and red disks showing which vertices we add to our vertex cover. How can we be sure the vertex cover we obtain will be as small as possible?

> **Before we take any steps, our (empty) subsolution can be extended to an optimal solution. Assume that, after $i \geq 0$ steps, our subsolution can be extended to an optimal solution S. Then we show that after $i + 1$ steps, our subsolution can be extended to an optimal solution S'. Therefore, by induction, we obtain an optimal solution.**[*]

Suppose that during our $(i + 1)$st step we consider an edge (u, v) and take v. If S includes v, then $S' = S$. If S does not include v, then it must contain u in order to cover that edge. Since all the other edges incident to u must already have been covered — because we consider (u, v) only when one of its endpoints is a leaf and then we take the other endpoint — removing u from S leaves only (u, v) uncovered. Therefore, $S' = \{v\} \cup (S\ \{u\})$ (that is, v added to S with u removed) is a vertex cover of the same size as S that extends our subsolution after $i + 1$ steps.

Another "BOOM" problem with easy and hard versions is Knapsack: suppose you're going on a hiking trip and you're deciding which of a set of items to take with you in a knapsack, with

[*]I hope you're getting the impression that I want you to remember this.

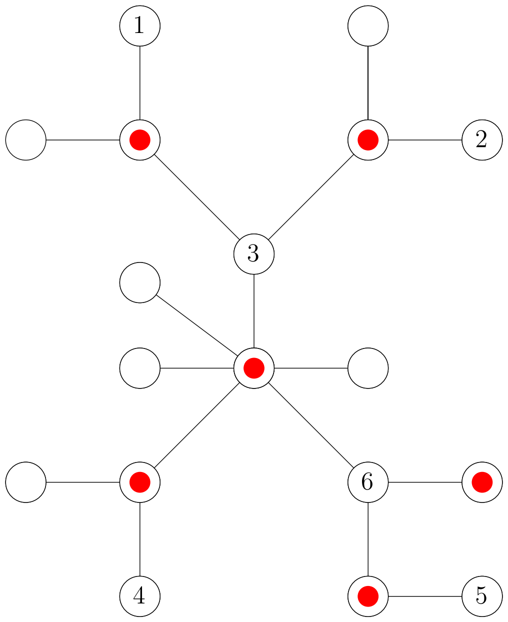


Figure 9.4: A minimum vertex cover of a tree.

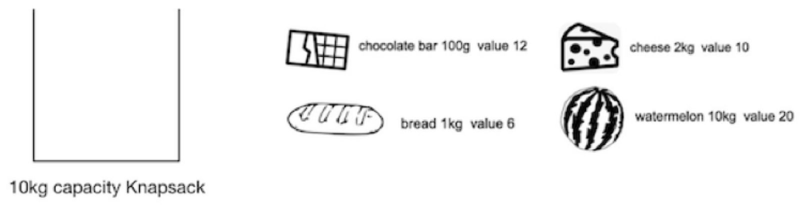
each item having a weight and a profit and the knapsack having a certain limit on the weight it can hold, called its *capacity*; you want to choose items such that you obtain the maximum possible profit without exceeding the knapsack's capacity.

The easy version of Knapsack is called Fractional Knapsack, and I think it's really only used as an example problem for teaching greedy algorithms. The normal, hard version of Knapsack is usually just called Knapsack, but sometimes also Binary Knapsack or 0-1 Knapsack to distinguish it from Fractional Knapsack.

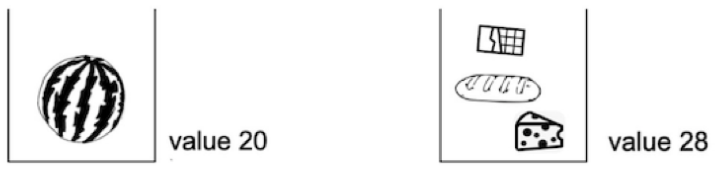
Binary Knapsack will come up again when we get to dynamic programming, and again when we get to NP-completeness. I'll show you an algorithm for Binary Knapsack that solves even the hard version optimally, and runs in time polynomial in the capacity — even though it's a "BOOM" (that is, NP-complete) problem. It was to prepare you for that apparent contradiction that I told you we usually measure running times in terms of the size of the paper it takes to write out the instance, and had you think about why factoring is used in crypto-systems even though we can factor a number n in $O(n)$ arithmetic operations.

For now, let's concentrate on Fractional Knapsack, which is called that because we can cut the items: if we cut an item and discard some fraction of it, then the profit of the remaining part is proportional to its original profit times the fraction that we keep. Figure 9.5 shows some of the figures from the scribe notes on this topic, which for some reason make me smile.

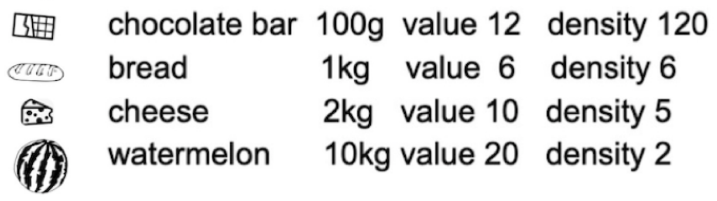
Taking the most profitable item (such as a giant watermelon) first could be a mistake since, as shown in the second part of Figure 9.5, it might fill the whole knapsack. Taking the smallest item first could also be a mistake. Since we can cut the items, the best approach is to order the items into non-increasing order by their profit-to-weight ratios — their *densities* — then take each item



"Empty knapsack with items can be taken"

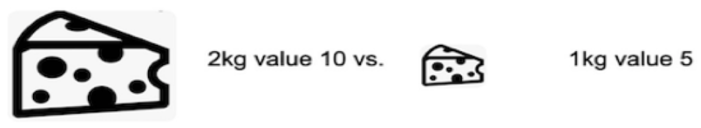


"A giant watermelon vs cheese+bread+chocolate bar"



	chocolate bar	100g	value 12	density 120
	bread	1kg	value 6	density 6
	cheese	2kg	value 10	density 5
	watermelon	10kg	value 20	density 2

"Each item with value density"



"cut cheese into half and get half value"

Figure 9.5: Some memorable figures from the 2020 scribe notes.

as long as it fits in the knapsack and, when we reach an item that won't fit in the knapsack, to cut it and take as much of it as we can. Why does this fill the knapsack as profitably as possible?

> **Before we take any steps, our (empty) subsolution can be extended to an optimal solution. Assume that, after $i \geq 0$ steps, our subsolution can be extended to an optimal solution S. Then we show that after $i + 1$ steps, our subsolution can be extended to an optimal solution S'. Therefore, by induction, we obtain an optimal solution.**[†]

Suppose that for our $(i + 1)$st step we take g grams of some item x and S includes g' grams of it. Since we always take the whole item when we can, and as much as possible when it won't all fit, $g \geq g'$. Suppose we take S and cut out $g - g'$ grams of whatever it includes instead of those last $g - g'$ grams of x, and fill the extra space up with x. The profit cannot go down, because $g - g'$ grams of x are at least as profitable as $g - g'$ grams of anything we consider after x, so we obtain a solution S' that is as profitable as S and extends our subsolution after $i + 1$ steps.

[†]Don't worry, this is the last time I'm going to write this. Of course, *you* should probably still write it for various assignments and exams. . .

Assignment 4

posted: 11.06.2021 **due: midnight 18.04.2021**

You can work in groups of up to three people. One group member should submit a copy of the solutions on Brightspace, with all members' names and banner numbers on it; the other group members should submit text files with all members' names and banner numbers (otherwise Brightspace won't let us assign them marks!). You may consult with other people but each group should understand the solutions: after discussions with people outside the groups, discard any notes and do something unrelated for an hour before writing up your solutions; it's a problem if no one in a group can explain one of their answers. For programming questions you should submit your code, which should compile and run correctly to receive full marks.

1. You have a week to complete an assignment with several questions, each worth the same number of marks. You don't want to spend more than h hours on the whole assignment and you can estimate accurately how many hours each question will take you. Give a greedy algorithm to decide which questions to answer. PROVE YOUR ALGORITHM CORRECT!

2. Your professor is training to run against his friend Simon, but he's not sure he can make it around his whole planned route in one go, so he's made a list of places where he can stop for a break, coffee, etc. (For example, if he starts at the shipyards and runs along the coast, he can stop at the Tim Horton's by the ferry terminal, then in the Salt Yard, then at one of the restaurants along the waterfront, then at the Garrison Brewery or Tomavinos by the Seaport Market, then at the entrance of Point Pleasant, then at the top of Arm Road in the park, etc etc.) Suppose he gives you this list, with the distance between each consecutive pair of potential pit stops, and the distance d he can run without stopping. Give a greedy algorithm that tells him where to stop such that 1) he never runs more that distance d without a break and 2) he makes the minimum number of stops. PROVE YOUR ALGORITHM CORRECT!

3. A *cross parsing* of a string $S[1..m]$ with respect to a string $T[1..n]$ is a partition of S into a minimum number of substrings each of which occurs in T. Suppose you are given an array $L[1..m]$ such that, for $1 \leq i \leq m$, the substring $S[i..i+L[i]-1]$ occurs in T but the substring $S[i..i+L[i]]$ doesn't. Give a greedy algorithm for computing a cross parsing of S with respect to T. PROVE YOUR ALGORITHM CORRECT!

4. According to `https://www150.statcan.gc.ca/t1/tbl1/en/tv.action?pid=1710000901`, the populations of Canada's provinces and territories in the first quarter of 2021 were as follows:

Newfoundland and Labrador	520,438
Prince Edward Island	159,819
Nova Scotia	979,449
New Brunswick	782,078
Quebec	8,575,944
Ontario	14,755,211
Manitoba	1,380,935
Saskatchewan	1,178,832
Alberta	4,436,258
British Columbia	5,153,039
Yukon	42,192
Northwest Territories	45,136
Nunavut	39,407

Suppose we choose a resident of Canada uniformly at random and let X be the province or territory where they live.

(a) Compute the entropy (in bits) of the random variable X.

(b) Compute $\sum_i p_i \lceil \lg(1/p_i) \rceil$, where p_i is the probability the resident of Canada lives in the ith province or territory listed above.

(c) Build a Huffman code for the probability distribution $p_1, \ldots, p_{13}$; what is its expected codeword length?

5. Suppose you have season passes for m train lines between n cities, with different expiry dates. A ticket lets you travel on the line between two cities as many times as you like, in either direction, from now until the ticket expires. How can you quickly determine the last date on which you will be able to reach any city from any other city using your passes?

(a) Give a solution with the union-find data structure that takes $O(m\,\alpha(m, n))$ time after you've sorted the passes by expiry date.

(b) Give a solution that colours and re-colours the cities, and takes $O(m)$ time after you've sorted the passes by expiry data.

You need not prove your solutions correct.

Midterm 1

posted: 9 am, 07.06.2021 **due: midnight, 11.06.2021**

You are **not** allowed to work in groups for the midterm. You can look in books and online, but you must not discuss the exam problems with anyone. If you don't understand a question, contact Travis or one of the TAs for an explanation (but no hints). All problems are weighted the same and, for problems broken down into subproblems, all subproblems are weighted the same.

1. For each cell (i, j) in the matrix below, write o, O, Θ, Ω or ω to indicate the relationship of the ith function on the left to the jth function along the top. If none of those relationships hold, leave the cell blank. Only the best answer possible will be considered correct (so writing O when the best answer is o doesn't count, for example). The cell $(1, 1)$ is filled in as an example: $n^{1/4} \in o(n^2)$, so that cell contains "o". **You need not explain your answers.**

	n^2	$((-1)^n + 1)n$	$3^{\lg n}$	$n^{\lg \lg n}$	$n \lg n$
$n^{1/4}$	o				
2^n					
$\lceil \lg n \rceil !$					
$n \sum_{j=1}^{n} (1/j)$					
$T(n) = 5T(n/4) + n^{3/2}$					

2. Assuming you have an $O(n)$-time algorithm to find a separator of a planar graph on n vertices — that is, a subset of at most $2\sqrt{n}$ vertices after whose removal all remaining connected components each consist of at most $2n/3$ vertices — give a divide-and-conquer algorithm to find a minimum vertex cover of a planar graph on n vertices, similar to our algorithm for colouring a planar graph. Your mark will partly depend on how fast your algorithm is (although there is not believed to exist a polynomial-time algorithm). **You need not analyze your algorithm nor prove it correct.**

3. Suppose we have a function that, given an unsorted sequence of n integers, in $O(n)$ time returns the (n/q)th, $(2n/q)$th, ..., $((q-1)n/q)$th smallest elements, called q-quantiles. Considering the time to compare elements to quantiles,
   (a) how quickly can we sort with this function when q is constant?
   (b) how quickly can we sort with this function when $q = \sqrt{n}$?
   (c) if we can choose q freely, how should we choose it to sort as quickly as possible with this function?

   You need not explain your answers.

   (10% bonus question) How fast can we sort if the function takes $O(n)$ time and separates the integers in the array into $\sqrt{n}$ bins such that the ith bin contains the $((i-1)\sqrt{n} + 1)$st through $(i\sqrt{n})$th smallest integers? (That is, the first bin contains the smallest $\sqrt{n}$ elements, the second bin contains the next $\sqrt{n}$ elements, etc.) **You need not explain your answer.**

Continued on Next Page!

4. Imagine you're planning a post-lockdown canoe trip with friends, but
   - people want to bring different amounts of equipment,
   - everyone wants to be in the same canoe as their equipment,
   - you can have only so much equipment in each canoe (all the canoes are the same, and consider only the weight of the equipment),
   - any one person's equipment fits in one canoe,
   - everyone wants to row (so you can have at most two people in each canoe).

   You have a list of how much equipment each person wants to take (in kilos), and you know how much fits in a canoe. For example, if there are 3 people going and they want to take 37 kg, 52 kg and 19 kg of equipment and a canoe can hold up to 60 kg of equipment (plus up to 2 people), then you need at least 2 canoes: you can put the first and third people and their $37 + 19 = 56$ kg of equipment in one and the second person and their 52 kg in the other.
   Give a greedy algorithm to find the minimum number of canoes you need AND GIVE A PROOF OF CORRECTNESS!

5. Give a greedy algorithm for Binary Knapsack that runs in $O(n \log n)$ time, where n is the number of items to consider, and achieves at least half the maximum profit when all the items have the same profit-to-weight ratio. Explain why your algorithm achieves this.

Part III

Dynamic Programming

Chapter 10

Edit Distance

My high school biology teacher, Mr Klages, wasn't very strict when it came to tests. If a student asked him to clarify a question, he would first explain what the question was asking and then, if the students was still confused, he'd also explain the steps in the reasoning leading to the answer, and eventually he'd often end up saying something like "So the answer's 5, isn't it?". One of my classmates, Ryan, had an odd sense of humour (and a lot of patience!) and thought it was fun to keep asking Mr Klages questions until he'd revealed the answers to all the questions on the test:

Ryan:	"Mr Klages, I don't understand Question 1."
Mr Klages:	"Well, Ryan, it's asking for the cell organelles; remember those from last week?"
Ryan:	"Not really."
	⋮
Mr Klages:	"So the last one is *Golgi body*, isn't it?"
Ryan:	"Right, thanks, *Golgi body*..." [writes]
Mr Klages:	"Ok, so now you're all set?"
Ryan:	"Yes, but I don't understand Question 2 either."

(I sympathize more with Mr Klages now that I've tried teaching; `http://phdcomics.com/comics/archive.php?comicid=5`.)

Let's suppose Mr Klages is testing Ryan's ability to compute the minimum cost of a path from the top left corner of the matrix in Figure 10.1 to the bottom right, according to the following rules:

- at each step we can move right one cell, down one cell, or diagonally right and down one cell;
- crossing a black arrow costs \$1;
- crossing a red arrow is free.

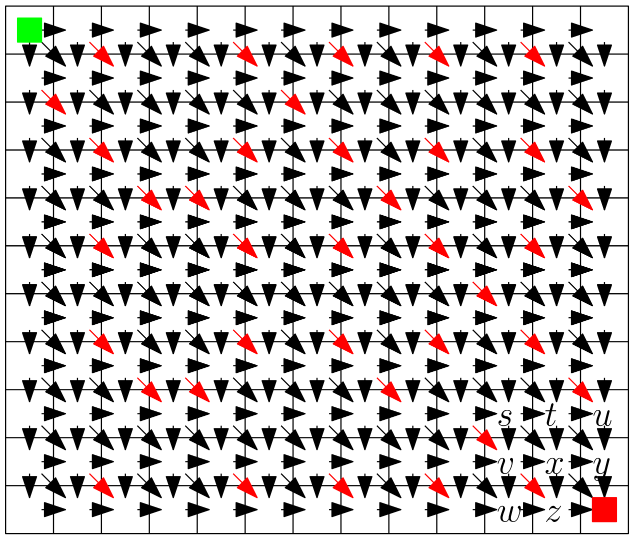


Figure 10.1: What is the minimum cost of a path from the top left to the bottom right if crossing a black arrow costs \$1 and crossing a red arrow is free?

Naturally, Ryan is going to say he doesn't know the minimum cost of a path from one corner to the other, and Mr Klages will say something like

> "Well, if I told you the minimum costs x, y and z of paths to the cells above and to the left, above, and to the left of the cell in the bottom right corner, respectively, then you could work out the minimum cost $\min(x+1, y+1, z+1)$ of a path to that cell, couldn't you?"

Ryan being Ryan, he'll say yes and ask what x, y and z, to which Mr Klages will answer something like

> "Well, if I told you the minimum costs s, t, u, v and w of paths to the cells above and to the left of those cells, then you could work out
>
> $$\begin{aligned} x &= \min(s+1, t+1, v+1)\,, \\ y &= \min(t+1, u+1, x+1)\,, \\ z &= \min(v, x+1, w+1)\,, \end{aligned}$$
>
> couldn't you? (Notice we can get from the cell with cost v to the cell with cost z for free, so we don't add a 1 to v when computing z!)"

What Mr Klages tells Ryan is essentially the following recurrence,

$$\begin{aligned} A[0,0] &= 0\,, \\ A[i,0] &= A[i-1,0]+1 \text{ for } i>0\,, \end{aligned}$$

$$
\begin{aligned}
A[0,j] &= A[0,j-1]+1 \text{ for } j>0\,,\\
A[i,j] &= \min\begin{pmatrix} A[i-1,j-1]+B_{i,j}, \\ A[i-1,j]+1, \\ A[i,j-1]+1 \end{pmatrix} \text{ for } i,j>0,
\end{aligned}
$$

where $B_{i,j}$ is an *indicator variable* equal to 1 if the arrow from $A[i-1,j-1]$ to $A[i,j]$ is black and 0 if it's red. Seeing a recurrence, computer scientists automatically think of recursion, and it's true Ryan could call the following program on the coordinates of the bottom right corner:

```
int cost(int i, int j) {
  if (i == 0 && j == 0) {
    return(0);
  } else if (i == 0) {
    return(cost(i, j - 1) + 1);
  } else if (j == 0) {
    return(cost(i - 1, j) + 1);
  } else {
    return(min(cost(i - 1, j - 1) + B[i, j],
      cost(i - 1, j) + 1, cost(i, j - 1) + 1));
  }
}
```

How long is that going to take, though? Notice we'll call `cost` for a cell once for every way to reach that cell from the bottom right corner using only moves left, up, and diagonally left and up. If we ignore diagonal moves, we'll get a lower bound on the number of calls to `cost` that's something like the number of binary strings on at most 22 bits containing at most 12 copies of 0 and at most 10 copies of 1, which is huge. Not even Mr Klages and Ryan are that patient!

We could *memoize* the answers to each call to `cost` — meaning we record them in an array so we don't need to recompute them — but instead of "unwrapping" the problem from the bottom right corner and backtracking, it's a bit cleaner to work from the top left corner. If we decide the top left corner has coordinates $(0,0)$ and the bottom right corner has coordinates (m,n) — there's a reason for this that we'll see later — then we can use the following iterative program:

```
int costDP(){

  A[0][0] = 0;

  for (int i = 1; i <= m; i++) {
    A[i][0] =  A[i - 1][0] + 1;
  }

  for (int j = 1; j <= n; j++) {
    A[0][j] = A[0][j - 1] + 1;
  }
  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      A[i][j] = min(A[i - 1][j - 1] + B[i][j],
        A[i - 1][j] + 1, A[i][j - 1] + 1);
    }
  }

  return(A[m][n]);
}
```

This is *dynamic programming*,* with $A[i, j]$ storing the minimum cost of reaching cell (i, j) in our original matrix. Of course the first and second `for` loops just set $A[i, 0] = i$ and $A[0, j] = j$, but it's useful to remember why they're doing that. The main work of the algorithm is the nested `for` loops, which take a total of $\Theta(mn)$ time. This is a fairly typical dynamic program so, before moving on to more serious problems, let's spend some time trying to optimize it.

First, notice this program finds only the cost of the minimum path, but not the path itself. We could fix that by creating a matrix $C[0..m, 0..n]$ and, whenever we fill in $A[i, j]$, we store in $C[i, j]$ from which direction we would arrive at $A[i, j]$ to achieve the minimum cost: if $A[i, j] = A[i-1, j-1] + B_{i,j}$ then we store $(i-1, j-1)$ in $C[i, j]$; otherwise, if $A[i, j] = A[i-1, j] + 1$ then we store $(i-1, j)$ in $C[i, j]$; otherwise, we store $(i, j-1)$ in $C[i, j]$. This lets us walk back along the path from $C[m, n]$.

Because we're filling in an $(m+1) \times (n+1)$ matrix (or two of them, if we want the path), we also use $\Theta(mn)$ space. Notice, however, that if we don't need the path, just the cost, then after we fill in $A[i+1, 0..n]$, we never use $A[i, 0..n]$ again and can discard it to save space. If m is much less than n, then we can fill A in column-major order instead of row-major order. This means we can use $\Theta(mn)$ time but only $\Theta(\min(m, n))$ space. If we want to recover the path itself while still using $o(mn)$ space, there are various tradeoffs we can use, but those are beyond the scope of this course.

An important optimization is called *banded dynamic programming*. Suppose you want to get from the top left corner to the bottom right corner but you have only \$7. Notice that reaching a cell (i, j) with $i > j + 7$ requires more than 7 moves straight down and thus costs more than \$7, while if $j > i + 7$ then reaching the cell requires more than 7 moves straight right and thus costs more than \$7. Therefore, we need only concern ourselves with the cells of the matrix in the band $A[i, j]$ with $|i - j| \leq 7$, which contains at most $(2 \cdot 7 + 1)n$ cells, as shown in Figure 10.2. In general, if we have a budget of k then we can find $A[m, n]$ in time $O(k \min(m, n))$ using this technique. This is important in practice because we often are only interested in the exact cost of an optimal solution when it is fairly small; as the cost grows, either we lose interest in the solution or we are more willing to accept some error.

Now that everyone's completely comfortable with finding a minimum-cost path through a matrix, we can discuss how to compute the edit distance between two strings $S[1..m]$ and $T[1..m]$,

*Incidentally, don't worry if you don't see why it's called that: `https://en.wikipedia.org/wiki/Dynamic_programming#History`.

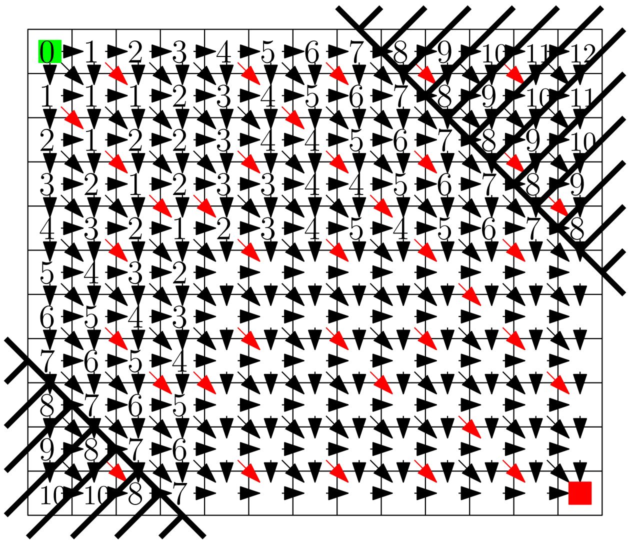


Figure 10.2: If we are interested only in solutions of cost 7 or less, we need fill in only the band $A[i, j]$ with $|i - j| \leq 7$.

which is the number of single-character insertions, deletions and substitutions needed to change S into T. For example, if $S =$ `AGATACATCA` and $T =$ `GATTAGATACAT`, then we can change S into T by prepending `GATT` and deleting the last two characters `AC`, so the edit distance is at most 6. How can we quickly tell if this is minimum?

Computing the edit distances between strings is a key task in bioinformatics in particular, since it lets us judge how similar pieces of DNA are and we can modify the computation to align pieces of DNA. For example, the alignment corresponding to the series of edits given above is

```
----AGATACATCA
GATTAGATACAT--
```

and another is

```
AGAT-ACAT-CA-
-GATTAGATACAT
```

with the second corresponding to only 5 edits to change S into T: deleting an `A`, inserting a `T`, changing a `C` to a `G`, inserting an `A`, and inserting a `T`.

It's worth noting that edit distance, also sometimes called Levenshtein distance, is a true distance measure: the distance between S and T is 0 if and only if they are equal, the distance from S to T is the distance from T to S, and the distance from S to T plus the distance from T to another string U is at least the distance from S to U (that is, the Triangle Inequality holds).

Let $M[i][j]$ be the minimum number of edits to turn a prefix $S[1..i]$ of S into a prefix $T[1..j]$ of T. Turning the empty prefix of S into the empty prefix of T requires no edits, so $M[0][0] = 0$. (It is convenient to index the characters of S and T starting from 1, even though it differs from how strings are usually indexed in code, so we have a row for the empty prefix of S and a column for the empty prefix of T.) Turning a prefix $S[1..i]$ of S into the empty prefix of T requires i edits, and turning the empty prefix of S into a prefix $T[1..j]$ of T requires j edits, so $M[i][0] = i$ and $M[0][j] = j$.

Suppose we have computed $M[i-1][j-1]$, $M[i-1][j]$ and $M[i][j-1]$. To turn $S[1..i]$ into $T[1..j]$, we can

- turn $S[1..i-1]$ into $T[1..j-1]$ with $M[i-1][j-1]$ edits and then turn $S[i]$ into $T[j]$ with 0 edits if they're already equal and 1 edit if not (so $S[i]$ is above $T[j]$ in the corresponding alignment);
- turn $S[1..i-1]$ into $T[1..j]$ with $M[i-1][j]$ edits and then delete $S[i]$ with 1 edit (so $S[i]$ is above a '`-`' in the alignment);
- turn $S[i]$ into $T[1..j-1]$ with $M[i][j-1]$ edits and then insert $T[j]$ with 1 edit (so $T[j]$ is below a '`-`' in the alignment).

In other words,

$$\begin{aligned}
M[0,0] &= 0\,, \\
M[i,0] &= M[i-1,0]+1 \text{ for } i>0\,, \\
M[0,j] &= M[0,j-1]+1 \text{ for } j>0\,, \\
M[i,j] &= \min\left(\begin{array}{l} M[i-1,j-1]+B_{i,j}, \\ M[i-1,j]+1, \\ M[i,j-1]+1 \end{array}\right) \text{ for } i,j>0,
\end{aligned}$$

where $B_{i,j}$ is an indicator variable equal to 0 if $S[i] = T[j]$ and 1 if they are not equal.

But this is the same recurrence we had for filling in A and finding the minimum-cost path through the matrix in Figure 10.1! In fact, if we consider Figure 10.3, we can see that the red arrows point into the cells (i,j) for which $S[i] = T[j]$. (The leftmost column and the top row don't have characters associated with them because they correspond to the empty prefixes of S and T.) So, much as we essentially rediscovered Huffman's algorithm while considering how to merge sorted lists, we've essentially derived the dynamic-programming algorithm for edit distance while thinking about paths through matrices. Once again, as my former supervisor would say, "I was *tricking* you!"

The technique of banded dynamic programming is so useful because we rarely try to compute the edit distance of long, dissimilar strings; most of the time, we're aligning human DNA with other human DNA, or at least two pieces of DNA from the same phylum.

We're almost done, but I'd like to mention two things. First, it's often useful to find the best alignment of a small string (such as a DNA read) against a substring of a long string (such as a chromosome). It doesn't make sense to use the algorithm we've developed so far, because the minimum number of edits to change a small string into a long string is at least the difference in lengths. If $S =$ `ACATA` and $T =$ `GATTAGATACAT`, for example, then we want the alignment

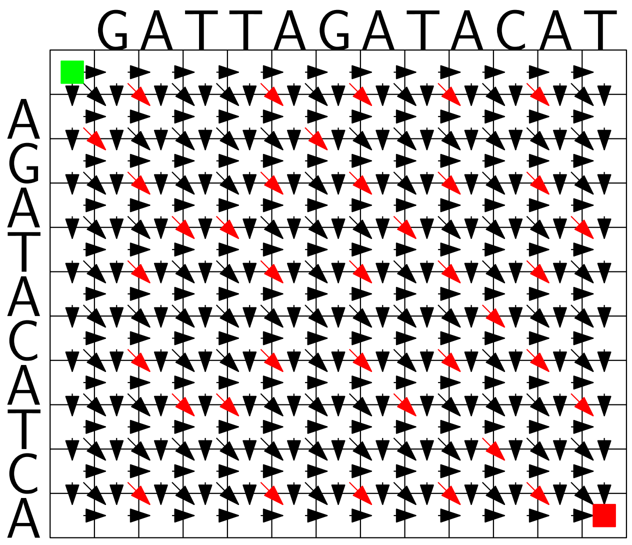


Figure 10.3: Computing the edit distance between S = `AGATACATCA` and T = `GATTAGATACAT` is equivalent to finding the cost of the cheapest path through the matrix from Figure 10.1. The red arrows point into cells (i, j) such that $S[i] = T[j]$.

```
----ACATA---
GATTAGATACAT
```

to correspond to only 1 edit — changing `C` to `G` — and not 8. To make the algorithm compute this modified cost, it is enough to change the costs to 0 for moving to the right along the top and bottom rows (in other words, changing the black arrows in the top and bottom rows to red). Whereas an alignment computed with the previous algorithm is called a *global* alignment, an alignment computed with this algorithm is called a *local* alignment.

The last thing I want to mention is a closely related problem, called LONGEST COMMON SUBSEQUENCE (LCS), for which we are asked to find the longest subsequence common to two strings. (Recall that the characters in a subsequence need not be consecutive; if they are then it is a sub-

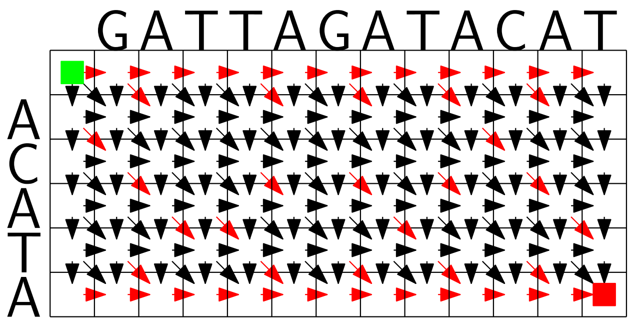


Figure 10.4: The matrix for computing the local alignment of `ACATA` and `GATTAGATACAT`.

string.) If we consider all the characters that are not edited when turning S into T, then they are a common subsequence. In other words, all the characters for which we cross red arrows while computing a global alignment are a common subsequence between the two strings. Therefore, if we change the recurrence so that crossing black arrows are free and we are paid \$1 for crossing red edges, we can compute an LCS in essentially the same way that we compute edit distance. Code for computing an LCS is shown below.

```
#include <stdio.h>

#define MAX(a, b) (a > b ? a : b)
#define MAX3(a, b, c) (MAX(a, b) > c ? MAX(a, b) : c)

int LCS(char *S, char *T, int m, int n) {
  int A[m + 1][n + 1];

  A[0][0] = 0;

  for (int i = 1; i <= m; i++) {
    A[i][0] = 0;
  }

  for (int j = 1; j <= n; j++) {
    A[0][j] = 0;
  }

  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      A[i][j] = MAX3(A[i - 1][j], A[i][j - 1],
        A[i - 1][j - 1] + (S[i - 1] == T[j - 1] ? 1 : 0));
    }
  }

  return(A[m][n]);
}

int main() {
  char *S = "AGATACATCA";
  char *T = "GATTAGATACAT";

  printf("The length of the LCS between %s and %s is %i.\n",
    S, T, LCS(S, T, 10, 12));
  return(0);
}
```

Chapter 11

Subset Sum and Knapsack

I've mentioned before that Knapsack is a "BOOM" problem but we can solve it in time polynomial in the capacity of the knapsack. More specifically, we can solve it in time polynomial in the capacity of the knapsack when measured in a unit such that capacity and all the items' weights are integers. This means we can't speed things up by a factor of 1000 just by measuring the capacity of the knapsack in kilos instead of grams.

Before we consider Knapsack, however, let's consider a simpler "BOOM" problem that can be solved by dynamic programming, called Subset Sum. For this problem, we are given a set $X = \{x_1, \ldots, x_m\}$ of integers and an integer target n, and asked if a subset of X sums to n. (I'm using m to denote the number of integers and n to denote the target because it will lead to us filling another $(m + 1) \times (n + 1)$ matrix.) For example, suppose $m = 6$ and $x_1, \ldots, x_6 = 2, 5, 8, 9, 11, 12$ and $n = 15$. It's pretty obvious the only solution is $2 + 5 + 8 = 15$, but it's not so obvious how to find that solution in a way that will still be reasonably efficient even for large instances.

The key step to designing a dynamic program is usually deciding what you'll store in each cell of your matrix. (I used to work with someone who would often propose solving open problems by "DP!", meaning dynamic programming, without giving any more details; that's only slightly more informative than saying "Math!".) For Subset Sum, the classic solution fills a Boolean matrix $A[0..m, 0..n]$ such that cell $A[i, j]$ stores true if a subset of $\{x_1, \ldots, x_i\}$ sums to *exactly* j, and false otherwise.*

It's easy to fill in $A[0..m, 0]$ and $A[0, 1..n]$, because the empty set sums to 0 so $A[i, 0] = \mathsf{true}$ for $i \geq 0$ and $A[0, j] = \mathsf{false}$ for $j > 0$. To see how to fill in $A[i, j]$ when we've filled in $A[i - 1, 0..n]$, consider that when trying to select a subset of $\{x_1, \ldots, x_i\}$ that sums to exactly j, we have only two choices of what to do with x_i:

*For simplicity, I'm assuming the x_is are positive; otherwise, we should fill in a matrix $A[0..i, s_-..s_+]$, where s_- is the sum of the negative x_is and s_+ is the sum of the positive x_is. Notice it's not as easy as offsetting each number by the same amount such that the smallest number is 0, since we don't know the size of the subset that sums to n, if there is one, and so we don't know how many times to add the offset to the target n.

- we can include it in the subset, in which case we are left with the subproblem of selecting a subset of $\{x_1, \ldots, x_{i-1}\}$ that sums to exactly $j - x_i$ — and there is one if and only if $A[i-1, j-x_i] = \mathsf{true}$;
- we can exclude it from the subset, in which case we are left with the subproblem of selecting a subset of $\{x_1, \ldots, x_{i-1}\}$ that sums to exactly j — and there is one if and only if $A[i-1, j] = \mathsf{true}$.

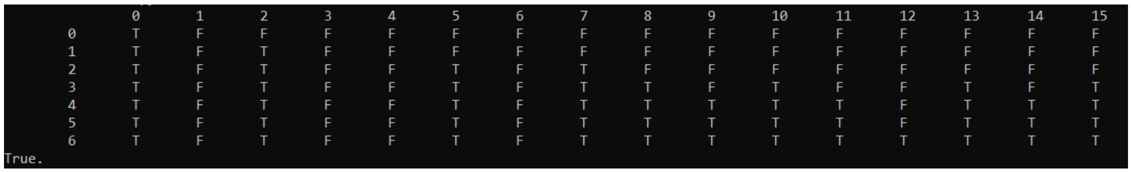
Therefore, we fill in the matrix with the recurrence

$$A[i,j] = \begin{cases} \mathsf{true} & \text{if } j = 0 \\ \mathsf{false} & \text{if } i = 0 \text{ and } j \neq 0 \\ A[i-1, j-x_i] \vee A[i-1, j] & \text{otherwise,} \end{cases}$$

treating $A[i-1][j-x_i]$ as false whenever $j - x_i < 0$. This takes $\Theta(mn)$ time (which is polynomial in m and n but not necessarily in the size of the input (X, n), since we need only $O(\lg n)$ bits to write n). The correctness of the algorithm follows from the correctness of the base cases, the recurrence, and induction.

Code to solve our example instance is shown below, with the matrix printed to `stderr`, and Figure 11.1 shows its output.

```
#include <stdio.h>

#define bool int
#define TRUE 1
#define FALSE 0

bool subsetSum(int *X, int m, int n) {
  bool A[m + 1][n + 1];

  for (int i = 0; i <= m; i++) {
    A[i][0] = TRUE;
  }

  for (int j = 1; j <= n; j++) {
    A[0][j] = FALSE;
  }

  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      A[i][j] = A[i - 1][j] ||
        ((j - X[i] >= 0) && A[i - 1][j - X[i]]);
    }
  }

  fprintf(stderr, "\t");
  for (int j = 0; j <= n; j++) {
```



```
        0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
    0   T   F   F   F   F   F   F   F   F   F   F   F   F   F   F   F
    1   T   F   T   F   F   F   F   F   F   F   F   F   F   F   F   F
    2   T   F   T   F   F   T   F   T   F   F   F   F   F   F   F   F
    3   T   F   T   F   F   T   F   T   T   F   T   F   F   T   F   T
    4   T   F   T   F   F   T   F   T   T   T   T   T   F   T   T   T
    5   T   F   T   F   F   T   F   T   T   T   T   T   F   T   T   T
    6   T   F   T   F   F   T   F   T   T   T   T   T   T   T   T   T
True.
```

Figure 11.1: The matrix for our example of Subset Sum.

```
    fprintf(stderr, "\t%i", j);
  }
  fprintf(stderr, "\n");

  for (int i = 0; i <= m; i++) {
    fprintf(stderr, "\t%i", i);
    for (int j = 0; j <= n; j++) {
      if (A[i][j]) {
        fprintf(stderr, "\tT");
      } else {
        fprintf(stderr, "\tF");
      }
    }
    fprintf(stderr, "\n");
  }

  return(A[m][n]);
}

int main() {
  int X[] = {0, 2, 5, 8, 9, 11, 12};
  // the leading 0 means we can index X from 1,
  // which is less confusing
  int m = 6;
  int n = 15;

  if (subsetSum(X, m, n)) {
    printf("True.\n");
  } else {
    printf("False.\n");
  }

  return(0);
}
```

Once you understand the dynamic-programming algorithm for Subset Sum, understanding the algorithm for Knapsack is not too difficult. Recall that for Knapsack we are given a set

$X = \{x_1, \ldots, x_m\}$ of (weight, profit) pairs, with $x_i = (w_i, p_i)$, and a capacity n, and asked to find the maximum profit of a subset of X with weight at most n.[†] Once again, I'm naming the parameters so that we fill in an $(m+1) \times (n+1)$ matrix; for that, we must also assume the weights and capacity have been scaled so they are all integers. For example, suppose $x_1 = (2, 1), x_2 = (3, 4), x_3 = (3, 3), x_4 = (4, 5)$ and $n = 8$, so the maximum profit achievable is 9 (with x_2 and x_4).

Now, we want $A[i, j]$ to store the maximum profit we can achieve with a subset of $x_1, \ldots, x_i$ with weight exactly j. Assuming the weights and profits are all positive, it's again easy to fill in $A[0..m, 0]$, since we can get no profit with no weight; for technical reasons that should be clearer in a moment, we set $A[0, 1..n]$ to $-\infty$ to indicate we cannot choose an empty subset with positive weight.

To see how to fill in $A[i, j]$ when we've filled in $A[i-1, 0..n]$, consider that when trying to select a subset of $\{x_1, \ldots, x_i\}$ with maximum profit whose weights sum to exactly j, we have only two choices of what to do with x_i:

- we can include it in the subset and receive profit p_i for it, in which case the maximum profit we can achieve from $\{x_1, \ldots, x_i\}$ is p_i plus the maximum profit we can achieve from a subset of $\{x_1, \ldots, x_{i-1}\}$ with weight exactly $j - w_i$;
- we can exclude it from the subset and no profit for it, in which case the maximum profit we can achieve from $\{x_1, \ldots, x_i\}$ is the maximum profit we can achieve from a subset of $\{x_1, \ldots, x_{i-1}\}$ with weight exactly j.

Therefore, we fill in the matrix with the recurrence

$$A[i, j] = \begin{cases} 0 & \text{if } j = 0 \\ -\infty & \text{if } i = 0 \text{ and } j \neq 0 \\ \max(p_i + A[i-1, j-w_i], A[i-1, j]) & \text{otherwise,} \end{cases}$$

treating $A[i-1][j-w_i]$ as $-\infty$ whenever $j - w_i < 0$. This takes $\Theta(mn)$ time (which is again polynomial in m and n but not necessarily in the size of the input (X, n)). Again, the correctness of the algorithm follows from the correctness of the base cases, the recurrence, and induction.

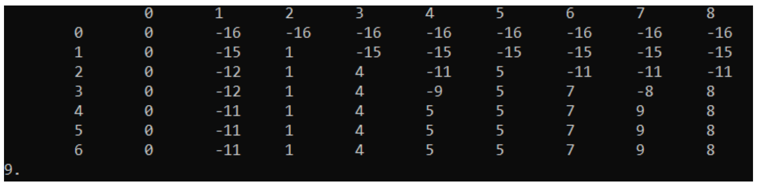
We put $-\infty$ in a cell $A[i, j]$ to indicate we cannot select a subset of $x_1, \ldots, x_i$ with weight j, and consider $A[i, j] = \infty$ for $j < 0$, because $-\infty$ will not change no matter what we add to it. We could also sum all the weights, multiply them by -1 and subtract 1 (as in the code below); or take the largest number, multiply it by $-m$ and subtract 1; etc.

Although it's simpler to fill in the matrix when we define $A[i, j]$ to be the maximum profit achievable from a subset of $\{x_1, \ldots, x_i\}$ with weight *exactly* j, we eventually want the maximum profit achievable with a subset of any weight up to the capacity, which is the maximum entry in the row $A[i, 0..n]$.

Code to solve our example instance is shown below, with the matrix printed to `stderr`, and Figure 11.2 shows its output.

[†]This version is an optimization problem; for the decision version, we need a target parameter t that is the minimum acceptable profit, so we should answer true if there is a subset with weight at most n and profit at least t, and false otherwise.

```
#include <stdio.h>

#define MAX(a, b) (a > b ? a : b)

int Knapsack(int *W, int *P, int m, int n) {
  int A[m + 1][n + 1];
  int sum = 0, max = 0;

  for (int i = 0; i <= m; i++) {
    A[i][0] = 0;
    sum += P[i];
  }

  for (int j = 1; j <= n; j++) {
    A[0][j] = - 1 * sum - 1;
  }

  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      A[i][j] = MAX(A[i - 1][j], P[i] +
        (j - W[i] >= 0 ? A[i - 1][j - W[i]] : - 1 * sum - 1));
    }
  }

  for (int j = 0; j <= n; j++) {
    max = MAX(A[m][j], max);
  }

  fprintf(stderr, "\t");
  for (int j = 0; j <= n; j++) {
    fprintf(stderr, "\t%i", j);
  }
  fprintf(stderr, "\n");

  for (int i = 0; i <= m; i++) {
    fprintf(stderr, "\t%i", i);
    for (int j = 0; j <= n; j++) {
      fprintf(stderr, "\t%i", A[i][j]);
    }
    fprintf(stderr, "\n");
  }

  return(max);
}
```



```
                0       1       2       3       4       5       6       7       8
        0       0       -16     -16     -16     -16     -16     -16     -16     -16
        1       0       -15     1       -15     -15     -15     -15     -15     -15
        2       0       -12     1       4       -11     5       -11     -11     -11
        3       0       -12     1       4       -9      5       7       -8      8
        4       0       -11     1       4       5       5       7       9       8
        5       0       -11     1       4       5       5       7       9       8
        6       0       -11     1       4       5       5       7       9       8
9.
```

Figure 11.2: The matrix for our example of Knapsack.

```
int main() {
  int W[] = {0, 2, 3, 3, 4};
  int P[] = {0, 1, 4, 3, 5};
  // the leading 0s means we can index P and W from 1,
  // which is less confusing
  int m = 6;
  int n = 8;

  printf("%i.\n", Knapsack(W, P, m, n));

  return(0);
}
```

There's a lot more to know about using dynamic programming to solve "BOOM" problems, but it's hard to discuss it reasonably when we're still using terms like "clink" and "BOOM" and it's probably beyond the scope of this course anyway. If you study algorithms in the future, you'll probably learn how to use dynamic programming to compute in truly polynomial time approximate answers to "BOOM" problems for which we can compute exact answers in time polynomial in the parameters (rather than the sizes of their binary representations).

It would be unfair to finish, however, without giving you an example of a "BOOM" problem we think is hard even if all its parameters are given in unary rather than binary (so "polynomial-time" means "polynomial in the sum of all the numbers"). You probably remember the question about canoes from the midterm, for which you had to partition a list of numbers into a minimum number of subsets such that

- every subset had cardinality at most 2,
- no subset summed to more than a given amount.

The restriction on the cardinality was justified because "everyone wants to row". What if people don't care about rowing and thus we can put 3 people in a canoe? More formally, given a set of n numbers, can we partition it into subsets of size exactly 3 such that every subset sums to the same amount?

This problem is called 3-Partition and, although we can obviously solve it by trying all possible $n^{n/3}$ partitions, it is believed there is no algorithm that runs in polynomial time even in the sum of the numbers (and if there is, then we can find Hamiltonian paths in polynomial time, etc).

Chapter 12

Optimal Binary Search Trees and Longest Increasing Subsequences

A binary search tree (BST) on n keys must have height $\Omega(\log n)$, so the worst-case time for a search is also $\Omega(\log n)$. If we know the frequency with which each key is sought, however, then we can arrange the tree (maintaining the order of the keys) such that the more frequent keys are higher, and thus reduce the average time for a search. For example, if the keys are `apple`, `banana`, `grape`, `kiwi`, `mango`, `pear` and `quince` and their frequencies are 5, 3, 4, 1, 1, 2, 1, then with the tree shown on the left in Figure 12.1 we use a total of

$$5 \cdot 3 + 3 \cdot 2 + 4 \cdot 3 + 1 \cdot 1 + 1 \cdot 3 + 2 \cdot 2 + 1 \cdot 3 = 44$$

comparisons for the 17 searches, while with the tree shown on the right we use only

$$5 \cdot 2 + 3 \cdot 3 + 4 \cdot 1 + 1 \cdot 4 + 1 \cdot 3 + 2 \cdot 2 + 1 \cdot 3 = 37$$

comparisons (assuming that, when we search for a key x, we compare it to the key stored at each vertex on the path from the root to the vertex storing x).

A BST is *optimal* with respect to a certain distribution over its keys if it minimizes the average number of comparisons during a search, which is called the *cost* of the tree. In our example, the cost of the tree on the left in Figure 12.1 is 44/17 while the cost of the one on the right is 37/17. We can use dynamic programming to build efficiently an optimal BST for a given distribution, because any subtree of an optimal BST is itself optimal. To see why, consider that the total number of comparisons we perform while searching in a BST T is the total number of comparisons we perform in its left subtree T_L, plus the total number of comparisons we perform in its right subtree T_R, plus the number of comparisons we perform at the root (which is the number of searches or, equivalently, the total of all the frequencies). If T_L or T_R are sub-optimal then we can rearrange them to reduce the number of comparisons we perform in them during searches for their keys, and thus reduce the cost of T overall.

Imagine Mr Klages asking Ryan to build an optimal BST:

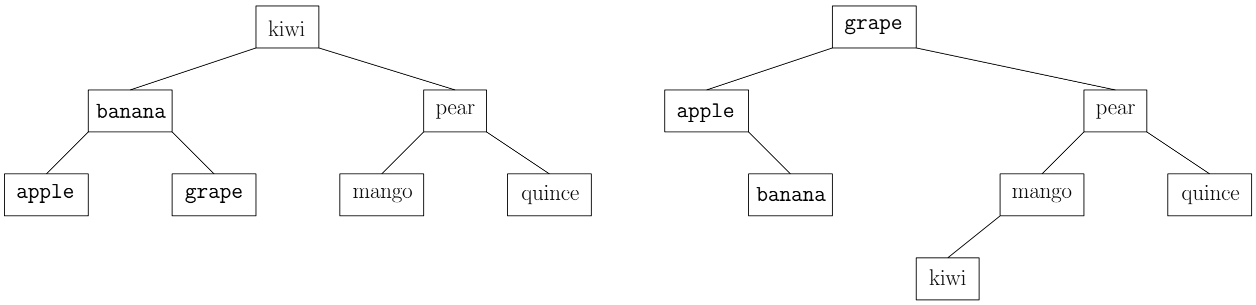


Figure 12.1: Two BSTs with keys `apple`, `banana`, `grape`, `kiwi`, `mango`, `pear` and `quince`. The tree on the left has better worst-case performance (3 comparisons versus 4 for a single search) but, if the frequencies with which the keys are sought are 5, 3, 4, 1, 1, 2, 1 then the tree on the right has better average-case performance (37/17 comparisons per search versus 44/17).

Ryan: “Mr Klages, I don’t know what the optimal BST is.”

Mr Klages: “Well, Ryan, one of the keys has to be at the root. If it’s `apple`,
then the optimal BST has an empty left subtree, `apple` at the root,
and a right subtree that’s an optimal BST for `banana` to `quince`.”

Ryan: “But I don’t know the optimal BST for `banana` to `quince`.”

Mr Klages: Well, again, one of the keys is at the root of that subtree. If it’s `banana`,
then the optimal subtree has an empty left subtree, `banana` at the root,
and a right subtree that an optimal BST for `grape` to `quince`.”

Ryan: “But I don’t know the optimal BST for `grape` to `quince`.”

⋮

Mr Klages: “So now you know what the optimal BST is, assuming the root stores `apple`.”

Ryan: “But what if it doesn’t? What if it stores `banana`?”

Mr Klages: “Well, then the left subtree contains only `apple`, the root stores `banana`,
and the right subtree is an optimal BST for `grape` to `quince`.”

Ryan: “But I don’t know the optimal BST for `grape` to `quince`.”

Travis: “Yes you do, he just told you! Pay attention, Ryan!
How am I in the same class as this bozo?!”*

It’s easy to build an optimal binary search tree for a single key — it’s simply one vertex, storing that key — and if we know the optimal BST for any set of up to i consecutive keys out of n, then we can compute an optimal BST for any set of up to $i + 1$ consecutive keys by considering all the possible roots and choosing the one that minimizes the total cost.

*It’s important to remember the differences between recursion and dynamic programming, to avoid using exponential time and annoying your classmates.

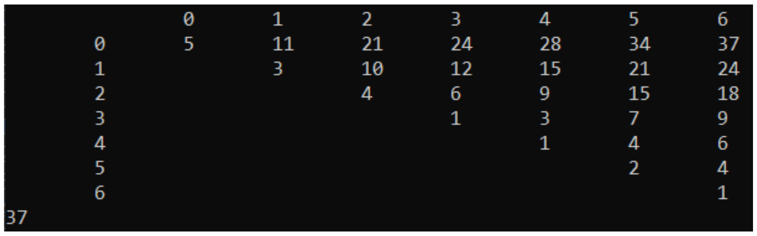
Suppose we are given keys $x_0, \ldots, x_{n-1}$ with frequencies $f_0, \ldots, f_{n-1}$. Let $A[i, j]$ be the number of comparisons we perform in an optimal BST for $x_i, \ldots, x_j$. Then our recurrence is

$$A[i,j] = \begin{cases} 0 & \text{if } i > j, \\ f_i & \text{if } i = j, \\ \min_{i \le r \le j}\{A[i, r-1] + A[r+1, j] + f_i + \cdots + f_j\} & \text{otherwise.} \end{cases}$$

We're still filling in a 2-dimensional matrix (or, more specifically, the top-right half, where $i \le j$), but now we cannot fill it in row-major or column-major order. Instead, we have to work along the diagonals, starting with the diagonal $j = i$ and then continuing with the diagonals $j = i + 1$, $j = i + 2$, etc — so the final answer is stored in $A[0, n-1]$.

Since $j - i + 1$ is the size of the tree for $A[i, j]$, in the code below we use a counter `size`. We can recover the tree itself by storing for each cell which choice of root minimized the cost of the corresponding subtree.

```
#include <stdio.h>

#define MIN(a, b) (a < b ? a : b)

int optBST(int *F, int n) {
  int A[n][n];

  for (int i = 0; i < n; i++) {
    A[i][i] = F[i];
  }

  for (int size = 2; size <= n; size++) {
    for (int i = 0; i < n - size + 1; i++) {
      int j = i + size - 1;
      A[i][j] = A[i + 1][j];
      for (int r = i + 1; r <= j - 1; r++) {
        A[i][j] = MIN(A[i][j], A[i][r - 1] + A[r + 1][j]);
      }
      A[i][j] = MIN(A[i][j], A[i][j - 1]);
      for (int k = i; k <= j; k++) {
        A[i][j] += F[k];
      }
    }
  }

  return(A[0][n - 1]);
}

int main() {
  int F[] = {5, 3, 4, 1, 1, 2, 1};
```



```
                0       1       2       3       4       5       6
        0       5       11      21      24      28      34      37
        1               3       10      12      15      21      24
        2                       4       6       9       15      18
        3                               1       3       7       9
        4                                       1       4       6
        5                                               2       4
        6                                                       1
37
```

Figure 12.2: The top-right half of the matrix we fill in for our example.

```
  printf("%i\n", optBST(F, 7));
  return(0);
}
```

If we add the following code before the `return` statement in the function `optBST`, we get the output shown in Figure 12.2:

```
  fprintf(stderr, "\t");
  for (int j = 0; j < n; j++) {
    fprintf(stderr, "\t%i", j);
  }
  fprintf(stderr, "\n");

  for (int i = 0; i < n; i++) {
    fprintf(stderr, "\t%i", i);
    for (int j = 0; j < n; j++) {
      fprintf(stderr, "\t");
      if (j >= i) {
        fprintf(stderr, "%i", A[i][j]);
      }
    }
    fprintf(stderr, "\n");
  }
```

As we can see from the output, the BST on the right in Figure 12.1 is indeed optimal.

This algorithm takes $\Theta(n^3)$ time because, to fill in each cell $A[i, j]$ with $j > i$, it checks $j - i + 1$ cells that have previously been filled in. (It also sums f_i through f_j, but this can be avoided.) Knuth showed that the best choice for the root of the tree with keys $x_i, \ldots, x_j$ is not to the left of the best choice for the root of the tree with keys $x_i, \ldots, x_{j-1}$ nor to the right of the best choice for the root of the tree with keys $x_{i+1}, \ldots, x_j$. In other words, adding keys to the right cannot make the best choice of root move left, and adding them to the left cannot make it move right.

Let $r_{i,j}$ be the best choice of root for the tree with keys $x_i, \ldots, x_j$. Then when taking advantage of Knuth's lemma, we use time proportional to

$$\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} (r_{i+1,j} - r_{i,j-1} + 1)\,.$$

In particular, when working out the best choices of roots for subtrees of size s, we use time proportional to

$$(r_{1,s-1} - r_{0,s-2}) + (r_{2,s} - r_{1,s-1}) + \cdots + (r_{n-s+1,n-1} - r_{n-s,n-2}) = r_{n-s+1,n-1} - r_{0,s-2} \leq n\,,$$

so overall we use $O(n^2)$ time.

There's an algorithm that runs in $O(n \log n)$ time, but it works only when all our searches are guaranteed to be successful. The dynamic programming algorithm we've been considering works even when the gaps between the keys are also assigned frequencies. For example, if the gap between `banana` and `grape` is assigned frequency 4, then we might search twice for `cherry` and once each for `date` and `fig`.

Another classic problem solvable with dynamic programming is finding the longest increasing subsequence (LIS) in a sequence of numbers.[†] Let's assume we're interested in subsequences that are strictly increasing, not just non-decreasing (although the same techniques work in both cases). For example, in the sequence $S = 5, 3, 2, 4, 1, 7, 9, 6, 8, 2, 7, 1, 5$, the subsequence $2, 4, 6, 8$ is increasing but the subsequence $2, 2, 7$ is not.

There is a nice dynamic-programming algorithm for LIS, which illustrates how the arrays for dynamic programs can be 1-dimensional (which it's easy to forget). If we define $S[-1] = -\infty$, $A[-1] = 0$ and $A[i]$ to be the length of the longest increasing subsequence ending at $S[i]$, for $0 \leq i \leq n-1$, then

$$A[i] = \max\{A[j] \;:\; -1 \leq j < i, S[j] < S[i]\} + 1\,.$$

Therefore, we can fill in A with the following code,

```
for (int i = 0; i < n; i++) {
  A[i] = 1;
  for (int j = 0; j < i; j++) {
    if (S[j] < S[i]) {
      A[i] = MAX(A[i], A[j] + 1);
    }
  }
}
```

We set $A[i]$ to 1 because we can always consider a single element as a subsequence. Since $A[n-1]$ is only the length of the longest subsequence ending exactly at $S[n-1]$, we must make a final pass to report the maximum value in $A[0..n-1]$.

[†]Recall we've already seen how to partition a sequence into the minimum number of increasing subsequences, using Supowit's greedy algorithm; I like to keep people on their toes by showing two similar-sounding problems with completely different solutions.

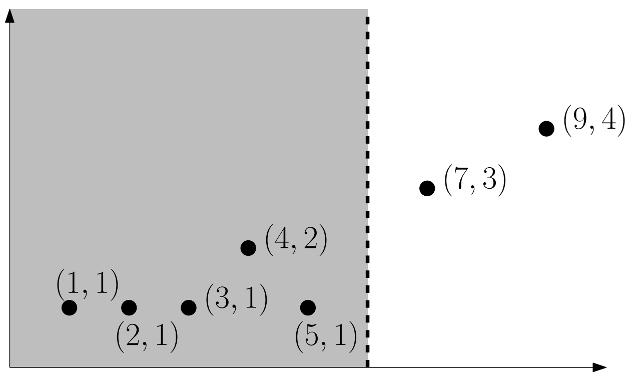


Figure 12.3: The query range $-\max(-\infty, 6)$ we use to compute $A[7]$ in our example.

This algorithm takes $\Theta(n^2)$ time but it can be improved to $O(n \log n)$ time fairly easily, using a *dynamic range-max data structure.*[‡] This data structure stores a set of 2-dimensional points and supports the following operations:

insert (x, y) adds the point (x, y) to the set;
delete (x, y) deletes the point (x, y) from the set;
range-max (x_1, x_2) returns the highest point in the half-open range $[x_1, x_2)$.

It's possible to adjust the definition of the range-max query so the query range is $[x_1, x_2]$, $(x_1, x_2]$ or (x_1, x_2); for finding LISs, the version above is the most convenient. The definition of the delete operation is given only for the sake of completeness, since we won't use it here.

Suppose we have a dynamic range-max data structure that supports all operations in $O(\log n)$ time when storing $O(n)$ points. If we start with the set of points empty and insert the point $(S[j], A[j])$ when we compute $A[j]$, then we can compute $A[i]$ in $O(\log n)$ time by using the query range $-\max(-\infty, S[i])$ to find the point $(S[j], A[j])$ with $S[j] < S[i]$ and $A[j]$ as large as possible. Figure 12.3 shows the points when processing $S[7] = 6$ in our example: the highest point in the query range $[-\infty, 6)$ is $(4, 2)$, so we set $A[7] = 3$ and insert the point $(6, 3)$; the grey area is meant to show the query range, although it should be a half-plane instead of a box, with the dashed line indicating that it does not include the points with $x = 6$.

We can implement a *static* range-max data structure with $O(\log n)$ time queries with an augmented BST, as shown in Figure 12.4. We store each x-coordinate as a key with the corresponding y-coordinate as satellite data; at each vertex in the tree, we store the point with the maximum y-coordinate stored in that vertex's subtree. Given a range $[x_1, x_2)$, we can search for x_1 and x_2 and find a collection of $O(\log n)$ single vertices and subtrees which store all the points with x-coordinates in the range $[x_1, x_2)$, and no other points.

For the single nodes, we check their y-coordinates and, for the subtrees, we check the point with the maximum y-coordinate stored in that subtree. This way, in $O(\log n)$ time, we can find

[‡]Actually, we don't need the full power of range-max queries for this particular problem — see `https://leetcode.com/problems/longest-increasing-subsequence/solution`, which is *much* clearer than it was last year — but I think you do for one of the questions on the assignment.

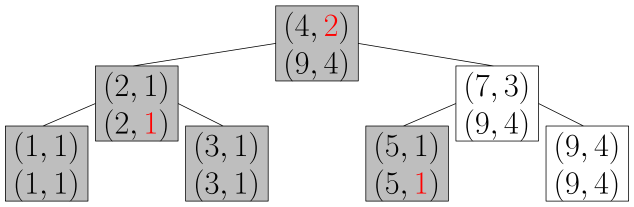


Figure 12.4: An augmented BST storing the points $(1,1),(2,1),(3,1),(4,2),(5,1),(7,3),(9,4)$ and supporting range-max queries. The shaded vertices are the ones with x-coordinates in the range $[-\infty,6)$ and the numbers in red are those we check for answering that query.

the point whose x-coordinate is in the range $[x_1,x_2)$ and whose y-coordinate is as large as possible. Figure 12.4 shows an augmented BST storing the points shown in Figure 12.3, with the top pair in each vertex being the x-coordinate stored as the key and the y-coordinate as satellite data, and the bottom pair being the highest point stored in the vertex's subtree. The shaded vertices are the ones with x-coordinates in $[-\infty,6)$, and we check the numbers shown in red for that query.

If you've seen AVL-trees then it shouldn't be too hard for you to figure out how to maintain the additional information in the tree when performing rotations — it's at the level of an interesting homework problem. Similarly, if you've seen red-black trees or $(2,4)$-trees or some other dynamic balanced binary search tree, then you should be able to figure out how to implement the updates to those such that they maintain the additional information. As a consequence, we obtain a *dynamic* range-max data structure, with insertions and deletions also taking logarithmic time.

If you haven't seen any dynamic balanced binary search trees, then I'm afraid there was a gaping hole in your second-year course on data structures (especially considering what university costs these days!). Unfortunately, we don't really have time to fix that in this course, which is full enough already. They're not that tricky, though, so an hour of YouTube tutorials should be more than enough.

Assignment 5

posted: 18.06.2021 **due: midnight 02.07.2021**

You can work in groups of up to three people. One group member should submit a copy of the solutions on Brightspace, with all members' names and banner numbers on it; the other group members should submit text files with all members' names and banner numbers (otherwise Brightspace won't let us assign them marks!). You may consult with other people but each group should understand the solutions: after discussions with people outside the groups, discard any notes and do something unrelated for an hour before writing up your solutions; it's a problem if no one in a group can explain one of their answers. For programming questions you should submit your code, which should compile and run correctly to receive full marks.

1. Write a program that takes two strings S and T and outputs an optimal alignment in $O(|S||T|)$ time, displayed as in the lecture notes. For example, if S = `AGATACATCA` and T = `GATTAGATACAT`, then an optimal alignment is

   ```
   AGAT-ACAT-CA-
   -GATTAGATACAT
   ```

   (I think). You need not analyze your algorithm nor prove it correct.

2. Write a linear-time dynamic-programming algorithm for `https://leetcode.com/problems/maximum-subarray` and EXPLAIN IT.

3. Suppose your professor has assigned a "profit" to each of several indivisible food items, expressing how much he likes each item. He's now filling his knapsack and trying to select items to maximize the total profit. The food items are light but bulky, so the key constraint now is the *volume* the knapsack can hold, rather than the *weight*. Even though the food items cannot be cut, they can be *squashed*, which reduces their volume by a factor of 2 — but also reduces their profit by a factor of 2 (since squashed food is not as appetizing).

   Write a dynamic-programming algorithm that runs in time polynomial in the number of items and the capacity of the knapsack (in litres) and tells your professor which food items to select and, of the selected ones, which ones to squash. You can assume the knapsack's capacity and the original volume in litres of each item are integers. Explain why your algorithm is correct.

4. Write pseudo-code — you don't have to code this — for an $O(n \log n)$-time algorithm that takes a sequence of n integers and finds the longest *slowly* increasing subsequence (LSIS), where an LSIS is a sequence in which each number after the first is larger than it's predecessor but not by more than 10. Explain why your algorithm is correct.

5. Modify the code in the lecture notes for building an optimal binary search tree such that it runs in $O(n^2)$ time instead of $O(n^3)$ time. You need not analyze your algorithm nor prove it correct.

Part IV

NP-Completeness

Chapter 13

Reductions

One of the coolest ideas in computer science is that many seemingly disparate problems are in some sense equivalent. In this lecture, we're going to prove (almost) the following theorem:

Theorem 9 *There exists a polynomial-time algorithm for problem $\mathcal{P}$ if and only if there exists a polynomial-time algorithm for problem $\mathcal{Q}$, for any $\mathcal{P}$ and $\mathcal{Q}$ in the set*

$$\{\,\textsc{Clique},\ \textsc{Independent Set},\ \textsc{Vertex Cover},\ 3\text{-}\textsc{Sat},\ \textsc{Sat},\ 3\text{-}\textsc{Col},\\ 4\text{-}\textsc{Col},\ 5\text{-}\textsc{Col},\ \textsc{Planar}\ 3\text{-}\textsc{Col},\ \textsc{Subset Sum},\ \textsc{Knapsack}\,\}\,.$$

To do this, we'll show that, given an instance X of problem $\mathcal{P}$, in time polynomial in the size of X we can turn X into an instance Y of problem $\mathcal{Q}$ such that Y is a positive instance if and only if X is a positive instance. For example, given a graph G on n vertices and an integer k, in time polynomial in n we can choose a set S of integers and an integer t such that some subset of S sums to t if and only if G has a clique of size k.[*]

Since we don't have a polynomial-time algorithm for deciding whether G has a clique of size k, we can't just work that out first and choose S and t accordingly. Instead, we turn G and k into S and t without knowing whether either is a positive instance. Doing that in time polynomial in n is called a *polynomial-time reduction* from Clique to Subset Sum.[†]

We're not going to see explicitly how to reduce each problem in the set above to each other problem, because we don't need to. Suppose we find one polynomial-time reduction `CliqueToSat` from Clique to Sat, and `SatTo3Sat` another from Sat to 3-Sat, and a third `3SatToSubsetSum` from 3-Sat to Subset Sum; then the composition of these reductions,

$$\texttt{3SatToSubsetSum}(\texttt{SatTo3Sat}(\texttt{CliqueToSat}(\cdot)))\,,$$

[*]I know I used X and n to denote the input to Subset Sum in a previous lecture; sorry!

[†]To remember that "$\mathcal{P}$ polytime reduces to $\mathcal{Q}$" means we can turn an instance of $\mathcal{P}$ into an instance of $\mathcal{Q}$ in polynomial time such that the latter instance is positive if and only if the former instance is positive, just think of "reduces" as "turns into". Someone pointed out that "we reduce" usually means "we make smaller" (and presumably easier), but here we if we can reduce $\mathcal{P}$ to $\mathcal{Q}$ then it means $\mathcal{Q}$ is not easier (by more than a polynomial factor) and may be harder. Sorry!

is a polynomial-time reduction from CLIQUE to SUBSET SUM. To see why, consider that

- reductions are functions (mapping instances of one problem to instances of another),
- polynomial-time functions have polynomial-sized outputs,
- polynomials are closed under composition (as well as under addition and multiplication, as we've already discussed).

This means, for example, that if `CliqueToSat` runs in time $f(n)$, where $f(x) = 3x^2 + 5x + 2$ is a polynomial and n is the size of `CliqueToSat`'s input, and `SatTo3Sat` runs in time $g(n')$, where $g(x) = 8x^3 + 14x^2 + 6x + 4$ is another polynomial and n' is the size of `SatTo3Sat`'s input, then `SatTo3Sat(CliqueToSat(`n`))` runs in time

$$\begin{aligned} g(f(n)) &= 8(3n^2 + 5n + 2)^3 + 14(3n^2 + 5n + 2)^2 + 6(3n^2 + 5n + 2) + 4 \\ &= 216n^6 + 1080n^5 + 2358n^4 + 2860n^3 + 2024n^2 + 790n + 136\,, \end{aligned}$$

which is just another polynomial.

You may have noticed we're only considering the *decision* versions of problems (which ask us whether there *exists* a clique of size k or a subset summing to t) rather than the *search* versions (which ask us to *find* a clique of size k or a subset summing to t). This is traditional, it makes the explanations simpler, and it doesn't really matter because, if we can determine in polynomial time whether any graph G has a clique of size k, for example, then in polynomial time we can find such a clique:

```
L = {}
for (each vertex v in G) {
  let G' be subgraph of G consisting of only v and its neighbours;
  if (G' has a clique of size k) {
    add v to L;
    G = G' minus v;
    k = k - 1;
  } else {
    G = G minus v;
  }
}
return(L)
```

The polynomial-time reductions we'll see first are:

- CLIQUE to INDEPENDENT SET (and vice versa),
- INDEPENDENT SET to VERTEX COVER (and vice versa),
- 3-SAT to INDEPENDENT SET,
- CLIQUE to SAT.

These are sufficient to prove that $\mathcal{P}$ has a polynomial-time algorithm if and only if $\mathcal{Q}$ does, for any $\mathcal{P}$ and $\mathcal{Q}$ in the set $\{$ CLIQUE, INDEPENDENT SET, VERTEX COVER $\}$.

3-Sat and Sat aren't in the set yet because, to add them, we need a polynomial time reduction from Sat to 3-Sat. There exists such a reduction, called the *Tseytin Transform*, but we won't see it until the next class; today, I'm asking you to take my word for it that 3-Sat polytime reduces to 3-Sat. Obviously 3-Sat reduces to Sat, because it's a special case of Sat.

Next, we'll see polynomial-time reductions from

- 3-Col to 4-Col (which we've seen before),
- 4-Col to 5-Col (which is the same),
- Planar 3-Col to 3-Col (which is trivial),
- 3-Col to Planar 3-Col,
- 3-Sat to 3-Col,
- 5-Col to Sat.

As long as you believe me about the Tseytin Transform, these expand the set to

$$\{\,\textsc{Clique}, \textsc{Independent Set}, \textsc{Vertex Cover}, \text{3-}\textsc{Sat}, \textsc{Sat}, \text{3-}\textsc{Col}, \text{4-}\textsc{Col}, \text{5-}\textsc{Col}, \textsc{Planar 3-Col}\,\}\,.$$

Finally, we'll see polynomial-time reductions from

- Subset Sum to Knapsack,
- 3-Sat to Subset Sum,
- Knapsack to Sat.

These complete our set and prove Theorem 9 (apart from the Tseytin Transform). There are actually now hundreds or even thousands of problems known to belong in the set — meaning they all polytime reduce to each other. Problems in this set are called *NP-complete*. "NP" stands for "non-deterministic polynomial time", and we'll discuss more in the next class what it means. Figure 13.1 shows our reductions, with the dashed arrow indicating the Tseytin Transform.

I feel a bit guilty showing you how to reduce problems to Sat, because no one ever does that. In the next lecture we'll see Cook's Theorem — yes, the same Cook as Toom-Cook multiplication — which is (essentially) a polynomial-time reduction from a problem that I'll call Input Checker to Sat. We'll see that any problem $\mathcal{P}$ for which we can check solutions in polynomial time, polytime reduces to Input Checker; Cook showed (essentially) that Input Checker polytime reduces to Sat; therefore, as long as we can check solution for $\mathcal{P}$ in polynomial time, $\mathcal{P}$ polytime reduces to Sat, and is said to be in NP.

Professors routinely dock marks because, when asked to prove that a problem $\mathcal{P}$ is NP-complete, students show that some NP-complete problem reduces to $\mathcal{P}$ but forget to show that $\mathcal{P}$ is in NP. There are problems for which we can't even check a solution in polynomial time, but I don't think they crop up all that often in real life, so I'm going to try not to be too pedantic about this.

I just realized I used $\mathcal{P}$ to denote a problem, when P stands for "deterministic polynomial-time" — the class of problems we can *solve* in polynomial time — so I'll just say that the question "Does P equal NP?" asks whether we can solve in polynomial time any problem for which we can check a

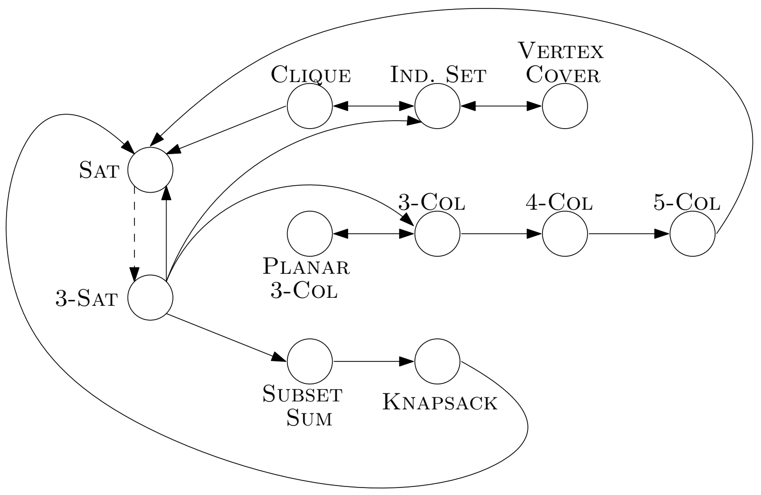


Figure 13.1: The polynomial-time reductions we prove and the Tseytin Transform (dashed) from Sat to 3-Sat.

solution in polynomial time. That sounds unlikely, and most people think it is, but after 50 years[‡] we don't have a proof either way — and finding a polynomial-time algorithm for *any* NP-complete problem would be enough to show P = NP.

Our first reductions are pretty trivial:

- a clique in a graph is a subgraph in which each pair of vertices have an edge between them, and an independent set is a subgraph in which no pair of vertices have an edge between them, so there is clique of size k in a graph G if and only if there is an independent set of size k in G's complement (that is, the graph on the same vertices as G in which there is an edge between two vertices u and v if and only if there is no edge between them in G);
- since the complement of an independent set (that is, the subset of vertices of the graph not in the independent set) is a vertex cover — to see why, consider that every edge must have at least one endpoint outside the independent set — there is an independent set of size k in a graph G on n vertices if and only if there is a vertex cover of size $n - k$.

The reduction from 3-Sat to Independent Set is slightly more complicated, though.

An instance of 3-Sat is a Boolean propositional formula in conjuctive normal form (CNF) in which each clause has exactly three literals (variables or their negations). The instance is positive if there is a satisfying truth assignment — that is, a way to set the variables to `true` or `false` such that the formula evaluates to `true`. For example,

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4)$$

is an instance of 3-Sat.

To turn an instance F of 3-Sat into an instance (G, k) of Independent Set, we

[‡]Cook's Theorem is from a paper published in the 1971 Symposium on Theory of Computing (STOC), and there was a panel discussion at STOC '21 last week to celebrate 50 years of the P vs. NP.

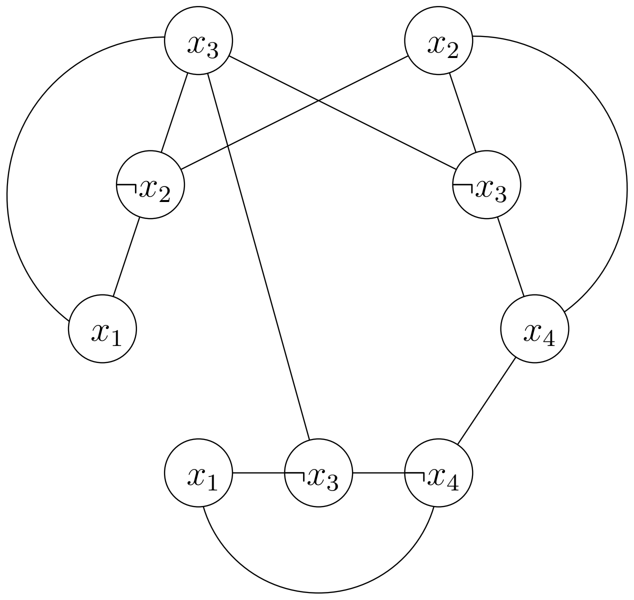


Figure 13.2: A graph that contains an independent set of size 3 if and only if $(x_1 \lor \neg x_2 \lor x_3) \land (x_2 \lor \neg x_3 \lor x_4) \land (x_1 \lor \neg x_3 \lor \neg x_4)$ has a satisfying truth assignment.

1. create a graph with a triangle for every clause and label each vertex with a literal from the clause;
2. add an edge between any vertices labelled x_i and $\neg x_i$, for any variable x_i.

This graph G has an independent set of size k, where k is the number of clauses in F, if and only if F has a satisfying truth assignment. To see why, suppose there is an independent set of size k. It can't contain two vertices from the same triangle, so we have one vertex from each triangle. None of the vertices in the independent set are labelled with a variable and its negation — otherwise there would be an edge between them — so it is possible to set all the literals labelling vertices in the independent set `true` simultaneously. Doing this makes every clause in F true, and thus F itself true. For example, we turn the instance

$$(x_1 \lor \neg x_2 \lor x_3) \land (x_2 \lor \neg x_3 \lor x_4) \land (x_1 \lor \neg x_3 \lor \neg x_4)$$

of 3-Sat into the instance $(G, 3)$ of Independent Set, where G is the graph shown in Figure 13.2.[§]

To reduce an instance (G, k) of Clique to an instance of Sat,

1. for every vertex v_i in G we create a variable x_i;
2. for every possible edge (v_i, v_j) that is *not* in G, we AND $\neg(x_i \land x_j)$ to our formula;
3. we AND to our formula a sub-formula that is satisfied if and only if at least k of the x_is are set to true.

Writing the sub-formula is the least obvious of these steps, but we can do it with something like a dynamic-programming recurrence, creating a variable $y_{i,j}$ that can be `true` in a satisfying truth

[§]It's more common to reduce 3-Sat to Clique, but I find the graph harder to draw and to understand then.

assignment if and only if at least j of $x_1, \ldots, x_i$ are set to `true` in that assignment, for $0 \le i \le n$ and $0 \le j \le k$.

We build the sub-formula from the following pieces, ANDed together:

- $y_{i,0} \Leftrightarrow$ `true` for all $0 \le i \le n$ (because at least 0 of $x_1, \ldots, x_i$ are always set to `true`),
- $y_{0,j} \Leftrightarrow$ `false` for $1 \le j \le k$ (because $j \ge 1$ of 0 variables can't be set to `true`),
- $y_{i,j} \Leftrightarrow (y_{i-1,j} \vee (y_{i-1,j-1} \wedge x_i))$ for $1 \le i \le n$ and $1 \le j \le k$ (because at least $j \ge 1$ of $x_1, \ldots, x_i$ are set to `true` if and only if and only if at least j of $x_1, \ldots, x_{i-1}$ are set to `true`, or at least $j-1$ of $x_1, \ldots, x_{i-1}$ are set to `true` and x_i is set to `true`),
- $y_{n,k}$ (because we want any satisfying truth assignment to have at least k of the x_is set to `true`).

(The symbol "$\Leftrightarrow$" means bidirectional implication or, equivalently, that the two sides have the same truth value — so "$a \Leftrightarrow b$" means "$(a \wedge b) \vee (\neg a \wedge \neg b)$".) The size of this formula is $O(nk)$, which is a polynomial, and it can be built in polynomial time.

Instead of writing out a long and detailed proof of correctness — because, as I said earlier, nobody reduces to SAT now — I'm just going to claim that "by inspection" our construction formula that is satisfiable if and only if G has a clique of size k. The idea is that there is a clique with vertices $v'_1, \ldots, v'_k$ in G if and only if there is there is a satisfying truth assignment with the corresponding literals $x'_1, \ldots, x'_k$ set to `true` (the other x_is can be set to `false`).

For example, if G is the graph shown in Figure 13.3 and $k = 4$, then our instance of SAT is

$$\begin{array}{ll}
 & \neg(x_3 \wedge x_4) \\
\wedge & \neg(x_3 \wedge x_5) \\
\wedge & (y_{0,0} \Leftrightarrow \texttt{true}) \\
\wedge & (y_{1,0} \Leftrightarrow \texttt{true}) \\
\wedge & (y_{2,0} \Leftrightarrow \texttt{true}) \\
\wedge & (y_{3,0} \Leftrightarrow \texttt{true}) \\
\wedge & (y_{4,0} \Leftrightarrow \texttt{true}) \\
\wedge & (y_{5,0} \Leftrightarrow \texttt{true}) \\
\wedge & (y_{0,1} \Leftrightarrow \texttt{false}) \\
\wedge & (y_{0,2} \Leftrightarrow \texttt{false}) \\
\wedge & (y_{0,3} \Leftrightarrow \texttt{false}) \\
\wedge & (y_{0,4} \Leftrightarrow \texttt{false}) \\
\wedge & (y_{1,1} \Leftrightarrow (y_{0,1} \vee (y_{0,0} \wedge x_1))) \\
\wedge & (y_{1,2} \Leftrightarrow (y_{0,2} \vee (y_{0,1} \wedge x_1))) \\
\wedge & (y_{1,3} \Leftrightarrow (y_{0,3} \vee (y_{0,2} \wedge x_1))) \\
\wedge & (y_{1,4} \Leftrightarrow (y_{0,4} \vee (y_{0,3} \wedge x_1))) \\
\wedge & (y_{2,1} \Leftrightarrow (y_{1,1} \vee (y_{1,0} \wedge x_2)))
\end{array}
\qquad
\begin{array}{ll}
\wedge & (y_{2,2} \Leftrightarrow (y_{1,2} \vee (y_{1,1} \wedge x_2))) \\
\wedge & (y_{2,3} \Leftrightarrow (y_{1,3} \vee (y_{1,2} \wedge x_2))) \\
\wedge & (y_{2,4} \Leftrightarrow (y_{1,4} \vee (y_{1,3} \wedge x_2))) \\
\wedge & (y_{3,1} \Leftrightarrow (y_{2,1} \vee (y_{2,0} \wedge x_3))) \\
\wedge & (y_{3,2} \Leftrightarrow (y_{2,2} \vee (y_{2,1} \wedge x_3))) \\
\wedge & (y_{3,3} \Leftrightarrow (y_{2,3} \vee (y_{2,2} \wedge x_3))) \\
\wedge & (y_{3,4} \Leftrightarrow (y_{2,4} \vee (y_{2,3} \wedge x_3))) \\
\wedge & (y_{4,1} \Leftrightarrow (y_{3,1} \vee (y_{3,0} \wedge x_4))) \\
\wedge & (y_{4,2} \Leftrightarrow (y_{3,2} \vee (y_{3,1} \wedge x_4))) \\
\wedge & (y_{4,3} \Leftrightarrow (y_{3,3} \vee (y_{3,2} \wedge x_4))) \\
\wedge & (y_{4,4} \Leftrightarrow (y_{3,4} \vee (y_{3,3} \wedge x_4))) \\
\wedge & (y_{5,1} \Leftrightarrow (y_{4,1} \vee (y_{4,0} \wedge x_5))) \\
\wedge & (y_{5,2} \Leftrightarrow (y_{4,2} \vee (y_{4,1} \wedge x_5))) \\
\wedge & (y_{5,3} \Leftrightarrow (y_{4,3} \vee (y_{4,2} \wedge x_5))) \\
\wedge & (y_{5,4} \Leftrightarrow (y_{4,4} \vee (y_{4,3} \wedge x_5))) \\
\wedge & y_{5,4}\,.
\end{array}$$

If I haven't made a mistake, this formula is satisfiable if and only if the graph in Figure 13.3 contains a clique of size 4. Of course, that graph does contain exactly one clique of size 4 (that is, with $v1, v_2, v_4, v_5$) and there should be exactly one satisfying true assignment: $x_1 = x_2 = x_4 = x_5 =$ `true`, $x_3 =$ `false`, and $y_{i,j}$ is `true` if at least j of $x_1, \ldots, x_i$ are `true` and `false` otherwise.

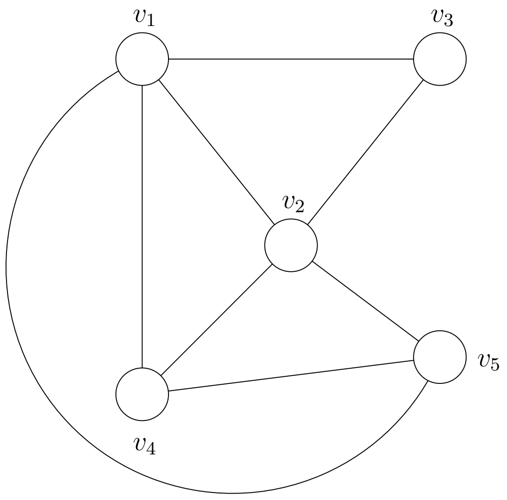


Figure 13.3: A graph that may contain a 4-clique.

We've already discussed how to reduce 3-COL to 4-COL: add a new vertex and put an edge between it and all the vertices already in the graph, so the new graph is 4-colourable if and only if the original graph is 3-colourable. We can reduce 4-COL to 5-COL the same way. Reducing PLANAR 3-COL to 3-COL is trivial, since the former is a special case of the latter: we needn't change the instance at all. To reduce 3-COL to PLANAR 3-COL, we use the *crossing gadget*[¶] shown in Figure 13.4 to replace edge crossings. It is left as an exercise for the reader to check that any 3-colouring of the crossing gadget has the same colour at the topmost and bottommost vertices, and the same colour at the leftmost and rightmost ones. Figure 13.5 shows how to use the crossing gadget to turn a non-planar graph G into a planar graph G' such that G' has a 3-colouring if and only if G does.

We reduce 3-SAT to 3-COL using clause gadgets, as we did when we reduced it to INDEPENDENT SET. We start with a triangle with vertices labelled `true`, `false` and `base`. For each variable x_i in the formula, we create a pair of vertices labelled x_i and $\neg x_i$, and make them two vertices in a triangle with the third vertex being `base`. If we 3-colour the graph so far, one of x_i and $\neg x_i$ will be the same colour as `true` and the other will be the same colour as `false`.

For each clause, we add a gadget as shown in Figure 13.6 (for $x_1 \vee \neg x_2 \vee x_3$). It is left as another exercise for the reader to check that the rightmost vertex can be the same colour as `true` if and only if at least one of the three leftmost vertices is. Finally, we add edges between the rightmost vertex in each clause gadget and `base` and `false`, so the only way the whole graph can be 3-coloured is if, for each clause, some vertex labelled with a literal in that clause is the same colour as `true` — meaning the formula is satisfiable. Figure 13.7 shows the graph for

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4)\,.$$

[¶]Reductions tend to be very structured, and the repeated substructures are called gadgets. For example, the triangles in our reduction from 3-SAT to INDEPENDENT SET are *clause gadgets*.

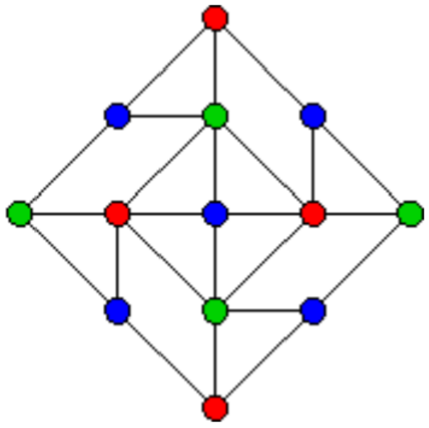


Figure 13.4: A gadget that can only be 3-coloured with the topmost and bottommost vertices the same colour, and the leftmost and rightmost ones the same colour.

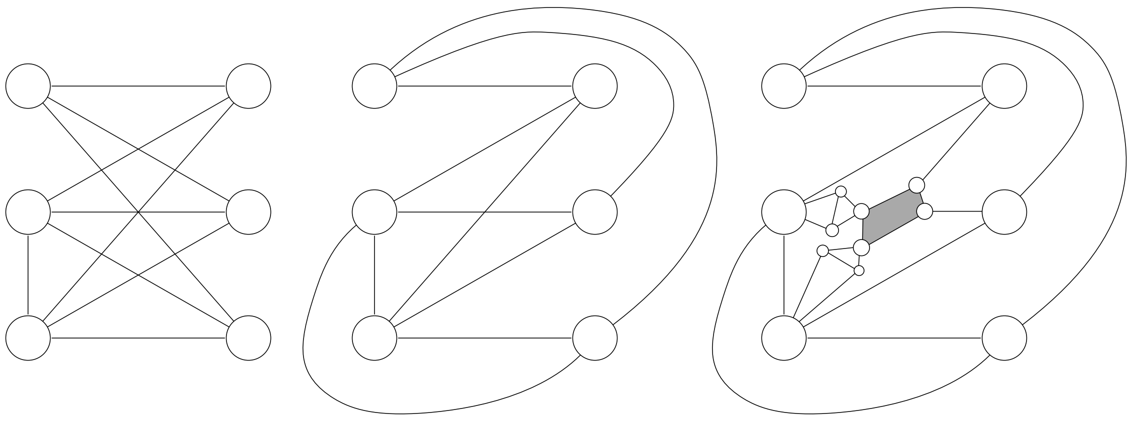


Figure 13.5: The graph G on the left can be re-embedded in the plane with only one edge-crossing **(center)** but not with 0. Nevertheless, the graph G' on the right — with the grey diamond representing a copy of the crossing gadget — can be 3-coloured if and only if G can be.

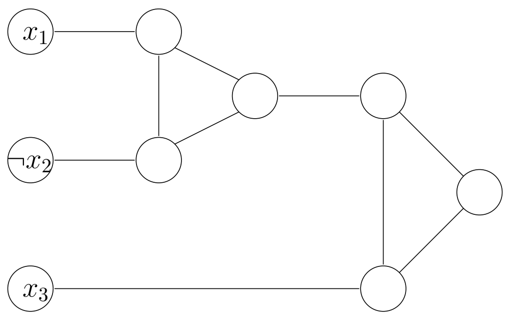


Figure 13.6: The rightmost vertex can be a colour in a 3-colouring if and only if at least one of the leftmost vertices is that colour.

Reducing 5-Col to 3-Sat — or k-Col to 3-Sat for any constant k — is somewhat easier than reducing Clique to 3-Sat, because now we needn't count anything. For every vertex v_i we create 5 variables, $\mathrm{red}_i, \mathrm{yellow}_i, \mathrm{green}_i, \mathrm{blue}_i, \mathrm{purple}_i$ and AND the following sub-formula to our formula, which is initially empty:

$$\begin{aligned} & (\mathrm{red}_i \lor \mathrm{yellow}_i \lor \mathrm{green}_i \lor \mathrm{blue}_i \lor \mathrm{purple}_i) \\ \land \quad & \neg(\mathrm{red}_i \land \mathrm{yellow}_i) \\ \land \quad & \neg(\mathrm{red}_i \land \mathrm{green}_i) \\ & \vdots \\ \land \quad & \neg(\mathrm{blue}_i \land \mathrm{purple}_i)\,. \end{aligned}$$

For each edge (v_i, v_j) in the graph, we AND the sub-formula

$$\begin{aligned} & \neg(\mathrm{red}_i \land \mathrm{red}_j) \\ \land \quad & \neg(\mathrm{yellow}_i \land \mathrm{yellow}_j) \\ \land \quad & \neg(\mathrm{green}_i \land \mathrm{green}_j) \\ \land \quad & \neg(\mathrm{blue}_i \land \mathrm{blue}_j) \\ \land \quad & \neg(\mathrm{purple}_i \land \mathrm{purple}_j)\,. \end{aligned}$$

to our formula. After this, the formula is satisfiable if and only if the graph is 5-colourable.

We have three reductions left to prove today: Subset Sum to Knapsack, 3-Sat to Subset Sum, and Knapsack to Sat. The first reduction is pretty obvious, since Subset Sum is something like a special case of Knapsack.

Formally, given an instance (S, t) of Subset Sum, where S is a set of integers and t is an integer, in time polynomial in the size of that instance we can turn it into an instance (P, p, w) of Knapsack, where P is a set of (profit, weight) pairs of integers and p and w are integers, such that there is a subset of S summing to exactly t if and only if there is a subset of pairs in P whose profits sum to at least p (the desired profit) and whose weights sum to at most w (the knapsack's capacity). To do this, we make every element x a pair (x, x) in P, and set both p and w to t. If

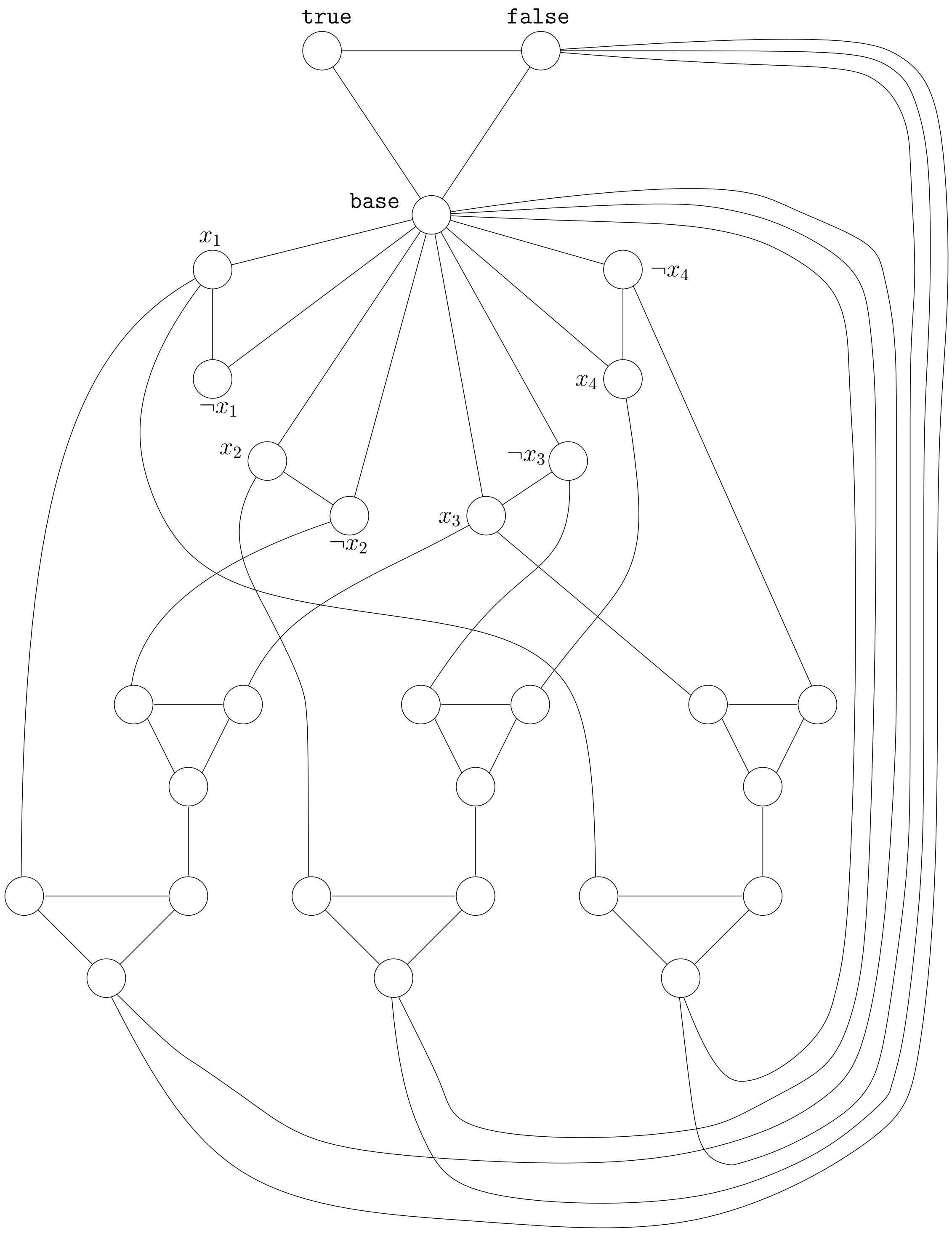


Figure 13.7: A graph that can be 3-coloured if and only if $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4)$ is satisfiable.

there is a subset S' of S summing to exactly t then

$$\sum_{x \in S'} x = t = p = w\,,$$

so choosing the corresponding pairs in P yields profit p with weight w. Conversely, if there is a subset P' of P whose profits sum to at least p and whose weights sum to at most w then

$$\sum_{x \in S'} x = t = p = w\,,$$

so choosing the corresponding elements of S yields a subset that sums to exactly t.

Reducing 3-SAT to SUBSET SUM is trickier. Given a Boolean 3-CNF formula F, we must choose a set S of integers and an integer t such that a subset of S sums to exactly t if and only if F has a satisfying truth assignment. Suppose F has n variables $x_1, \ldots, x_n$ and m clauses. We first build a $2(n+m) \times (n+m)$ grid of decimal digits (although we'll only use digits 0, 1 and 3), with the ith row being the ith integer in S.

For each variable x_i in F, we have two rows in our grid:

- one row has a 1 in the ith column and a 1 in the $(n+j)$th column if the jth clause contains the positive literal x_i, for $1 \leq j \leq m$, and 0s in all the other columns;
- the other row has a 1 in the ith column and a 1 in the $(n+j)$th column if the jth clause contains the negative literal $\neg x_i$, for $1 \leq j \leq m$, and 0s in all the other columns.

For the j clause in F, for $1 \leq j \leq m$, we also have two rows in our grid:

- one row has a 1 in the $(n+j)$th column and 0s everywhere else;
- the other row has a 2 in the $(n+j)$th column and 0s everywhere else.[‖]

We set t to be the number that starts with n copies of 1, followed by m copies of 4. Figure 13.8 shows the grid for

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4)\,.$$

Suppose a subset S' of S sums to exactly t. For the ith digit of the sum to be 1, for $1 \leq i \leq n$, S' must contain the number for the positive literal x_i or the negative literal $\neg x_i$, but not both. For the $(n+j)$th digit to be 4, for $1 \leq j \leq m$, S' must contain at least one of the numbers for a literal that makes the jth clause `true`. (If S' contains exactly one such number, it also contains both "slack" numbers for that clause, to bring the sum for the $(n+j)$th column up to 4; if it contains exactly two such numbers, it contains only the slack number with a 2 in the $(n+j)$th position; if it contains three such numbers, it contains only the slack number with a 1 in the $(n+j)$th position.) Therefore, there exists a satisfying truth assignment for F.

Now suppose there is a satisfying truth assignment for F, and consider the sum of the rows corresponding to literals that are `true` in that assignment. This number will start with n copies of

[‖]Some people put a 1 instead of a 2 in this row, but then rows are duplicates so we don't get a *multiset* of numbers instead of a *set*.

literal x_1	1	0	0	0	1	0	1
literal $\neg x_1$	1	0	0	0	0	0	0
literal x_2	0	1	0	0	0	1	0
literal $\neg x_2$	0	1	0	0	1	0	0
literal x_3	0	0	1	0	1	0	0
literal $\neg x_3$	0	0	1	0	0	1	1
literal x_4	0	0	0	1	0	1	0
literal $\neg x_4$	0	0	0	1	0	0	1
1st clause	0	0	0	0	1	0	0
1st clause	0	0	0	0	2	0	0
2nd clause	0	0	0	0	0	1	0
2nd clause	0	0	0	0	0	2	0
3rd clause	0	0	0	0	0	0	1
3rd clause	0	0	0	0	0	0	2

Figure 13.8: A set of integers with a subset that sums to 1111444 if and only if $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4)$ is satisfiable.

1, and end with m digits that are between 1 and 3. If the $(n+j)$th digit of the sum is 1, we add the two numbers with 0s everywhere else but 1 and 2 in the $(n+j)$th position, respectively; if that digit is 2, we add only the number with 0s everywhere else but 2 in the $(n+j)$th positon; if that digit is 3, we add only the number with 0s everywhere else but 1 in the $(n+j)$th position. This way, the final sum starts with n copies of 1 and ends with m copies of 4. Therefore, S has a subset that sums to exactly t.

Probably the hardest reduction we'll see today is from KNAPSACK to SAT. You'll probably see such a reduction only in this class, because it's quite painful and it's existence is implied by Cook's Theorem anyway, so normally there's no reason to talk about it. We're going to talk about it now for the insight I hope it will give you into Cook's Theorem and how that works (and to ensure no one ever again says this course is easier than Norbert's Principles of Programming Languages!).

Suppose we're given a set P of n pairs of (profit, weight) pairs of integers and integers p and w, and in time polynomial in n we want to build a formula F such that F is satisfiable if and only if there is a subset P' of P whose profits sum to at least p and whose weights sum to at most w.

For example, consider the instance of KNAPSACK we solved by dynamic programming in Chapter 11 — that is, $P = (1,2),(4,3),(3,3),(5,4)$[**] and $w = 8$ — with $p = 8$. Remember, the dynamic-programming algorithm we devised in Chapter 11 doesn't run in time polynomial in n, just time polynomial in n plus the capacity of the knapsack!

We start by creating a variable x_i for the ith pair (p_i, w_i), for $1 \leq i \leq n$. Let m be the number of bits in the binary representation of maximum of the sum of all the profits, the sum all the weights, p and w. For $1 \leq i \leq n$ and $1 \leq j \leq m$, if the jth bit of p_i is 1 then we AND $(p_{i,j} \Leftrightarrow x_i)$ to our formula, and if that bit is 0 we AND $(p_{i,j} \Leftrightarrow \texttt{false})$ to it; if the jth bit of w_i is 1 then we AND

[**]I seem to have reversed the order of the profits and the weights since Chapter 11; sorry again!

$$\begin{array}{ccccc}
(p_{1,1} \Leftrightarrow \texttt{false}) & \wedge(p_{1,2} \Leftrightarrow \texttt{false}) & \wedge(p_{1,3} \Leftrightarrow \texttt{false}) & \wedge(p_{1,4} \Leftrightarrow \texttt{false}) & \wedge(p_{1,5} \Leftrightarrow x_1) \\
\wedge(p_{2,1} \Leftrightarrow \texttt{false}) & \wedge(p_{2,2} \Leftrightarrow \texttt{false}) & \wedge(p_{2,3} \Leftrightarrow x_2) & \wedge(p_{2,4} \Leftrightarrow \texttt{false}) & \wedge(p_{2,5} \Leftrightarrow \texttt{false}) \\
\wedge(p_{3,1} \Leftrightarrow \texttt{false}) & \wedge(p_{3,2} \Leftrightarrow \texttt{false}) & \wedge(p_{3,3} \Leftrightarrow \texttt{false}) & \wedge(p_{3,4} \Leftrightarrow x_3) & \wedge(p_{3,5} \Leftrightarrow x_3) \\
\wedge(p_{4,1} \Leftrightarrow \texttt{false}) & \wedge(p_{4,2} \Leftrightarrow \texttt{false}) & \wedge(p_{4,3} \Leftrightarrow x_4) & \wedge(p_{4,4} \Leftrightarrow \texttt{false}) & \wedge(p_{4,5} \Leftrightarrow x_4) \\
\\
\wedge(w_{1,1} \Leftrightarrow \texttt{false}) & \wedge(w_{1,2} \Leftrightarrow \texttt{false}) & \wedge(w_{1,3} \Leftrightarrow \texttt{false}) & \wedge(w_{1,4} \Leftrightarrow x_1) & \wedge(w_{1,5} \Leftrightarrow \texttt{false}) \\
\wedge(w_{2,1} \Leftrightarrow \texttt{false}) & \wedge(w_{2,2} \Leftrightarrow \texttt{false}) & \wedge(w_{2,3} \Leftrightarrow \texttt{false}) & \wedge(w_{2,4} \Leftrightarrow x_2) & \wedge(w_{2,5} \Leftrightarrow x_2) \\
\wedge(w_{3,1} \Leftrightarrow \texttt{false}) & \wedge(w_{3,2} \Leftrightarrow \texttt{false}) & \wedge(w_{3,3} \Leftrightarrow \texttt{false}) & \wedge(w_{3,4} \Leftrightarrow x_3) & \wedge(w_{3,5} \Leftrightarrow x_3) \\
\wedge(w_{4,1} \Leftrightarrow \texttt{false}) & \wedge(w_{4,2} \Leftrightarrow \texttt{false}) & \wedge(w_{4,3} \Leftrightarrow x_4) & \wedge(w_{4,4} \Leftrightarrow \texttt{false}) & \wedge(w_{4,5} \Leftrightarrow \texttt{false})
\end{array}$$

Figure 13.9: The subformula we get after the first step of turning our example instance of KNAPSACK into an instance of SAT.

$(w_{i,j} \Leftrightarrow x_i)$ to our formula, and if that bit is 0 then we AND $(w_{i,j} \Leftrightarrow \texttt{false})$ to it. Figure 13.9 shows the subformula we get after this first step.

We now add variables $a_1, \ldots, a_m$ and $b_1, \ldots, b_m$, and subformulas saying the following:

- $a_1, \ldots, a_m$ are the bits of the binary number obtained by summing the n binary numbers whose bits are $p_{1,1}, \ldots, p_{1,m}$ and $p_{2,1}, \ldots, p_{2,m}$ and $p_{3,1}, \ldots, p_{3,m}$, etc.;
- $b_1, \ldots, b_m$ are the bits of the binary number obtained by summing the n binary numbers whose bits are $w_{1,1}, \ldots, w_{1,m}$ and $w_{2,1}, \ldots, w_{2,m}$ and $w_{3,1}, \ldots, w_{3,m}$, etc.;
- the binary number with bits $a_1, \ldots, a_m$ is at least as large as p;
- the binary number with bits $b_1, \ldots, b_m$ is at most as large as w.

I'm sure you could figure these subformulas out for yourselves if you needed to (since I made you study circuits for addition for the second assignment).[††] The result is an instance of SAT that is satisfiable if and only if you can assign truth values to $x_1, \ldots, x_n$ such that the p_is for the x_is that are true sum to at least p, and the w_is for the x_is that are true sum to at most w — in other words, if and only if there is a solution for the instance of KNAPSACK we started with.

Actually, that wasn't really all that bad, was it? Maybe I should worry about Principles of Programming Languages being harder than this course after all. On the bright side, now we're probably ready for Cook's Theorem.

[††]You might wonder about trying to avoid subformulas for addition by having a variable for each possible sum, something like we had a variable for every vertex and colour in the reduction from 3-COL to SAT. That can work, but the number of variables isn't polynomial in n, only in n plus the capacity of the knapsack.

Chapter 14

Cook's Theorem

Turing introduced what we now call Turing Machines (TMs) in order to reason about what computers can and, more importantly, *cannot* do. To see why it's harder to design robust models for proving lower bounds than for proving upper bounds, consider that if Turing had included in his definition of TMs some seemingly-reasonable assumptions about speed and memory, then by now we'd quite likely have broken those assumptions, in which case all his upper-bound proofs would still hold but his lower-bound proofs wouldn't. It's remarkable that he described TMs in the 1930s and, despite nearly a century of amazing technological advances, we still accept today that what cannot be computed on TMs cannot be computed at all — not then, not now and not ever.

If you've seen the movie *Hidden Figures* then you should know that in the early days of NASA, calculators were people who did calculations. Similarly, Turing started by considering *human* computation and had to argued it could be modelled by TM, before he could even start reasoning about the limits of TMs.[*] He was convincing enough that people accepted TMs as a universal model of computation more readily than they had accepted, say, Church's lambda calculus. I'll spare you those arguments, however, because you've grown up with computers and you'll probably believe me that if something can't be computed by any `C` program, for example, then we have good evidence (at least enough for this course) that it's incomputable.

A TM consists of a finite-state control and a tape with cells that we assume stretches off to infinity in both directions. At each step, the TM can

- read a symbol in the cell the control is currently over,
- write a symbol in the cell the control is currently over (overwriting whatever's there),
- move the tape one cell to the right or left,
- change its state.

You might think that's a silly model because

[*]Turing was very thorough: when proposing the Turing Test for artificial intelligence, he thought about whether it could be confounded by extra-sensory perception (ESP), because he couldn't categorically rule out the possibility ESP existed.

- finite-state machines are a really weak model of computation (they can't even recognize all strings of the form $a^n b^n$),
- we don't actually have infinite tapes.

On the other hand,

- any stand-alone computer is a finite-state machine,
- if you can't do something with an infinite tape then you can't do it with a finite tape either (so assuming the tape is infinite makes lower bounds *stronger*),
- no one has ever thought of a model that can compute something TMs can't (given enough time),
- theoretical computer scientists will make fun of you if you don't know what a TM is.

You should know about TMs but teaching them properly would take at least another full course, so I'm going to claim they're equivalent to something more familiar. Suppose your computer has 32 GB of RAM and 1 TB of disk space and you buy a tape-drive (yes, these are still used) and a sequence of 21 tape cassettes that you label $-10, \ldots, 10$, and attach the drive to your computer with cassette 0 in it. If your computer tries to move past the beginning of the current cassette, you replace it with the preceding cassette, wound to the end; if it tries to move past the end of the current cassette, you replace it with the next tape, rewound to the beginning; if you get to the beginning of the first cassette or to the end of the last cassette, then you go out and buy more cassettes and add them to the beginning or end of your sequence, numbered appropriately.

Your computer is essentially a $2^{1032000000003}$-state finite-state machine (the last digit is a 3 because a byte has 8 bits), so your computer and tape-drive and cassettes are together essentially a TM. Therefore, if no TM can compute something, then neither can your computer hooked up to a tape-drive, regardless of how many cassettes you buy. To prove the converse — that is, if your computer hooked up to a tape-drive with an infinite supply of cassettes can't compute something, then neither can any TM — we need the idea of a *universal TM*. This is a TM M_1 that, when started with a description of any other TM M_2 and an input X on its tape, simulates M_2 on X. People compete to see who can design the smallest universal TM, and by now I think they're down to a half-dozen states or so. Since we can easily implement such a universal TM on your computer as a `switch` statement with a half-dozen or so `case`s, if your computer hooked up to a tape-drive can't compute something, then neither can any TM. This means we don't have to worry about your computer not being able to compute something now because it doesn't have enough RAM, and that something becoming computable in a few years when memories are bigger. In fact, you could have saved nearly all the money you spent on RAM and your hard-drive and bought only enough memory to simulate a universal TM, as long as you don't mind polynomial slowdowns.

The *Church-Turing Hypothesis* claims that any decision problem that is *effectively* computable is computable on a TM. Here "effectively" means we can do it in a finite amount of time. What's called the *Strong Church-Turing Hypothesis* (which came after Church and Turing, actually) claims that any decision problem that is *efficiently* computable is computable in polynomial time on a deterministic TM. Here "efficiently" means we can do it in a reasonable amount of time. A *deterministic* TM is one whose finite-state control is deterministic, whereas a *non-deterministic* TM is one whose finite-state control is non-deterministic. Since a universal TM need not be more than a polynomial factor slower than the TM it's simulating, and your computer is deterministic, the

Strong Church-Turing Hypothesis implies that any decision problem that is efficiently computable is computable in polynomial time on your computer hooked up to a tape-drive (and now you need only polynomially many cassettes).

The complexity class P is the set of decision problems we can solve in polynomial time on a deterministic TM — so the class of efficiently computable problems, according to the Strong Church-Turing Hypothesis — and the class NP is the set of decision problems we can solve in polynomial time on a non-deterministic TM. *Never* say NP stands for "not polynomial"! That would be wrong, because P is obviously a subset of NP (and, although we strongly believe it's a proper subset, they could be equal).

A problem is called *NP-hard* if any problem in NP can be reduced to it in polynomial time. A problem is called *NP-complete* if

- it is in NP,
- it is NP-hard.

It's been shown that a problem is in NP if and only if we can *check* a solution to it in polynomial time on a *deterministic* TM.

The quintessential NP-complete problem to decide whether a given non-deterministic TM of size s halts in $f(s)$ time for some fixed polynomial f. In other words, the following statements are equivalent:

- there is a polynomial-time algorithm (that runs on a deterministic TM) for this problem,
- P = NP,
- for every problem for which we can *check* a solution in polynomial time (on a deterministic TM), we can *find* a solution in polynomial time (on a deterministic TM).

The "P vs. NP" problem asks if these statements are true.

That quintessential problem isn't very natural, however, and it's not easy to reduce it to many natural problems in the way we reduced 3-SAT to INDEPENDENT SET and 3-COL, for example. The goal of today's lecture is to prove Cook's Theorem, that SAT is NP-complete; thus, by the Tseytin Transform (which we'll see at the end of the day), 3-SAT is also NP-complete. It follows from our reductions in the last class that CLIQUE, INDEPENDENT SET, VERTEX COVER, 3-COL, etc., are all NP-hard and thus, because solutions to them can be checked in polynomial time, NP-complete. If there's a polynomial-time algorithm for one of them, there's a polynomial-time algorithm for all of them (and everything else in NP).

I made you do all those reductions to SAT because

- until we've proven Cook's Theorem, showing that a problem is in NP doesn't immediately imply it reduces to SAT,
- it's good practice for proving Cook's Theorem,
- it probably builds character.

After today, though, you'll never need to reduce anything to Sat again[†] since, by Cook's Theorem, showing you can check a solution in polynomial time will imply that such a reduction exists.

Theorem 10 (Cook's Theorem) Sat *is NP-complete.*

Incidentally, I should note here that Cook's Theorem is sometimes called the Cook-Levin Theorem, since Leonid Levin proved a version of it (that Circuit Sat is NP-complete) around the same time as Cook. It's not always called that because he gave lectures on his proof but didn't publish it, and he was working in the USSR during the Cold War so it took several years for his results to become known in the West, by which time Cook's Theorem was already famous.

We'll start by reducing the quintessential problem to the problem Input Checker, which uses more familiar terms:

> For the problem Input Checker, we are given a `C` program consisting of n characters and asked if there is an input that makes it output `true` in at most n steps (on your computer hooked up to a tape-drive).

Once we've done that, we won't need to talk about TMs again until the end of the course (when we discuss computability). After that reduction, we'll briefly consider how powerful Input Checker is, before reducing it to Sat. Finally, we'll cover the Tseytin Transform for reducing Sat to 3-Sat.

To see that Input Checker is in NP, consider that if we are given an input to a `C` program of length n then we can check in polynomial time whether the program outputs `true` in at most n steps: we just run the program. To see that Input Checker is NP-hard, suppose we are given a non-deterministic TM M of size s and we want to decide if M halts in time $f(s)$ for some fixed polynomial f. We can write a `C` program that takes as input a list of $f(s)$ transitions (where a transition says what M does in a step), checks they describe a valid computation of M leading to it halting and, if so, outputs `true`. This program will take $g(f(s))$ steps for some fixed polynomial g, and we can pad the program with comments until its length n is at least $g(f(s))$.

To understand how powerful Input Checker is, consider that even if you hadn't just read about TMs and NP, you could still easily reduce anything in NP (that is, any problem for which you can check a solution in polynomial time) to it. Consider Knapsack, for example: given (P, p, w), in polynomial time we can easily write a `C` program that, given a supposed solution consisting of a list of (profit, weight) pairs,

- checks those pairs are in P,
- checks their profits sum to at least p,
- checks their weights sum to at most w,
- outputs `true` if these three check are confirmed.

We can work out a polynomial upper bound on the number of steps this program takes and pad the program with comments until it has that length. There is an input that makes this program output

[†]Except for Midterm 2, and unless you ever need to solve constraint satisfaction problems (CSPs), which are often handled by reducing them to instances of Sat and feeding those to industrial Sat-solvers such as Z3.

`true` if and only if (P, p, w) is a `yes` instance of KNAPSACK. Think about some other problems we looked at in the last lecture and how you can reduce them directly to INPUT CHECKER.

Suppose we're given an instance C of INPUT CHECKER with n characters and in time polynomial in n we want to write an instance F of SAT such that F has a satisfying truth assignment if and only if C outputs `true` within n steps. First of all, since reading input takes time, we can assume C reads at most n bytes ($8n$ bits) of input. Since declaring variables and `malloc`ing memory takes time, we can also assume C doesn't use more than n bytes ($8n$ bits) of memory. We won't distinguish between the levels of the memory hierarchy — cache, main memory, disk, tape — so, if n bytes is more than your 32 GB of RAM and your 1 TB of disk and your computer has to fall back on the tape-drive, we'll still talk about "memory".

We create a variable x_i for $1 \leq i \leq 8n$ (one for each bit of input C could read), and a variable $y_{i,j}$ for $0 \leq i \leq 8n$ and $1 \leq j \leq n$ (corresponding to the ith bit of memory C could use, at the jth time step). Let's assume 32 of the memory bits are an output buffer, so if C sets them to

```
01110100011100100111010101100101
```

(that is, 116, 114, 117, 101, which are the ASCII codes for `t`, `r`, `u` and `e`), then C outputs `true`. We can create a variable z, write a subformula saying z is `true` if and only if the output buffer holds `true` at some point during C's first n steps, and AND z to that subformula (so any satisfying truth assignment for the subformula has to make z `true`).

We want to write our instance F of SAT such that, for any truth assignment that satisfies F and any input to C of at most $8n$ bits and any initial setting of the $8n$ bits of your computer's memory that C will use, if

- the truth assignment sets x_i to `true` if and only if the ith bit of the input to C is 1,
- the truth assignment sets $y_{i,0}$ to `true` if and only if the ith bit of the memory C uses is initially set to 1 (although well-written programs shouldn't rely on the initial contents of the memory they use!),

then, for $1 \leq i \leq 8n$ and $1 \leq j \leq n$, the truth assignment sets $y_{i,j}$ to `true` if and only if the ith bit of memory is set to 1 during the jth time step. In other words, we want F to model C's execution.

Let's assume that, for any truth assignment that satisfies F and any input to C of at most $8n$ bits and any initial setting of the $8n$ bits of your computer's memory that C will use and some j with $0 \leq j < n$, if

- the truth assignment sets x_i to `true` if and only if the ith bit of the input to C is 1,
- the truth assignment sets $y_{i,j}$ to `true` if and only if the ith bit of the memory C uses is set to 1 at time step j.

How can we add subformulas to F to guarantee that the truth assignment sets $y_{i,j+1}$ to `true` if and only if the ith bit of the memory C uses is set to 1 at time step $j + 1$? If we can do this for any j then, by induction, F has a satisfying truth assignment if and only if some input to C makes it output `true` in at most n steps.

If your CPU isn't "looking at" the ith bit during the jth step (that bit isn't in the CPUs registers and it's not being written to by the memory bus or hard-drive head or tape head during

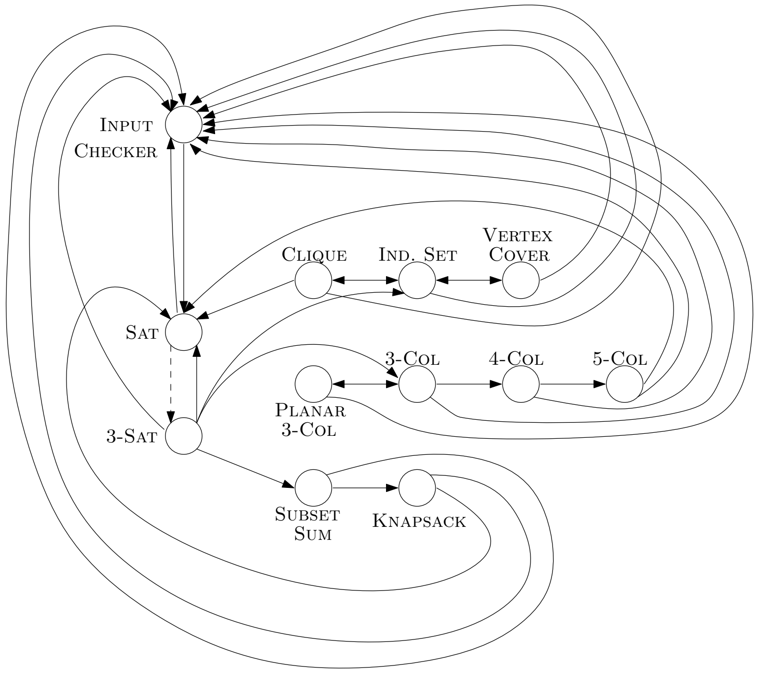


Figure 14.1: The polynomial-time reductions we proved in the previous lecture, the ones to INPUT CHECKER, the one from INPUT CHECKER to SAT, and the Tseytin Transform (dashed) from SAT to 3-SAT.

the jth step), then the value of the ith bit shouldn't change between the jth and $(j+1)$st steps. It's fairly easy to add subformulas expressing those constraints.

We can cheat here and observe that, since your computer is a finite-state machine (except for its tape-drive), we can simply use a subformula with $2^{1032000000003}$ variables — which is just a (big!) constant — and something like that length to capture its transition function. If you don't like that, then it's possible to consider your CPU as circuits (adders, for example) and write subformulas modelling how they behave. *The details are left as an exercise for the reader.* Either way, the subformulas will be pretty big, but only polynomial in n. In polynomial time, we get an instance F of SAT that is satisfiable if and only if C outputs `true` in at most n steps. In other words, INPUT CHECKER polytime reduces to SAT — so we've proven Cook's Theorem.

Adding INPUT CHECKER to Figure 13.1, we get Figure 14.1, with everything in NP polytime reducing to INPUT CHECKER and INPUT CHECKER polytime reducing to SAT, with only the Tseytin Transform missing to have SAT polytime reduce to 3-SAT, which polytime reduces to INDEPENDENT SET, 3-COL and SUBSET SUM.

The Tseytin Transform is pretty simple: given a Boolean formula F, we

1. rewrite F using only AND ($\wedge$), OR ($\vee$) and NOT ($\neg$) gates;
2. parenthesize F so that all operations are either unary (NOT) or binary (AND and OR);
3. initialize a new formula F' to be empty;

4. add to F' a CNF-subformula with at most 3 literals in each clause, saying a new variable is equal to the outcome of each operation in F;
5. add to F' the clause containing only the new variable equal to the outcome of the last operation;
6. pad each clause to have exactly 3 literals, by duplicating literals.

For Step 4, we use the following identities:

$$\begin{aligned}
(c \Leftrightarrow (a \wedge b)) &\quad\Leftrightarrow\quad ((\neg a \vee \neg b \vee c) \wedge (a \vee \neg c) \wedge (b \vee \neg c))\,, \\
(c \Leftrightarrow (a \vee b)) &\quad\Leftrightarrow\quad ((a \vee b \vee \neg c) \wedge (\neg a \vee c) \wedge (\neg b \vee c))\,, \\
(c \Leftrightarrow (\neg a)) &\quad\Leftrightarrow\quad ((a \vee \neg c) \wedge (\neg a \vee c))\,.
\end{aligned}$$

For example, if F after Steps 1 and 2 is

$$((\neg(x_1 \wedge x_2)) \vee (x_1 \wedge (x_2 \wedge (\neg x_3))))\,,$$

then after Step 5 F' is

$$\begin{aligned}
& (\neg x_1 \vee \neg x_2 \vee y_1) \wedge (x_1 \vee \neg y_1) \wedge (x_2 \vee \neg y_1) \\
\wedge\ & (y_1 \vee \neg y_2) \wedge (\neg y_1 \vee y_2) \\
\wedge\ & (x_3 \vee \neg y_3) \wedge (\neg x_3 \vee y_3) \\
\wedge\ & (\neg x_2 \vee \neg y_3 \vee y_4) \wedge (x_2 \vee \neg y_4) \wedge (y_3 \vee \neg y_4) \\
\wedge\ & (\neg x_1 \vee \neg y_4 \vee y_5) \wedge (x_1 \vee \neg y_5) \wedge (y_4 \vee \neg y_5) \\
\wedge\ & (\neg y_2 \vee \neg y_5 \vee y_6) \wedge (y_2 \vee \neg y_6) \wedge (y_5 \vee \neg y_6) \\
\wedge\ & y_6\,.
\end{aligned}$$

There is a *lot* more to know about complexity theory, but I think we've met the learning objectives for this course. At the moment, I think most computer scientists (at least, the ones who think about this kind of thing) believe in the Strong Church-Turing Hypothesis and that P is not equal to NP. Although we haven't proven P and NP are different, despite 50 years of trying, we know (by a result called Ladner's Theorem) that if they are different, then there are problems in NP that are neither in P nor NP-complete. Many people think factoring and graph isomorphism might be such problems, for example.

Interestingly, we have a polynomial-time algorithm, called Shor's algorithm, for factoring on quantum computers,[‡] so *if* factoring really isn't in P and *if* we can get a quantum computer actually running Shor's algorithm, then it'll disprove the Strong Church-Turing Hypothesis (but not the Church-Turing Hypothesis). As far as I know, however, no one has shown an algorithm for an NP-complete problem that runs in polynomial-time even on a quantum computer.

Of course, if P = NP, then everything in NP has a polynomial-time algorithm on a normal computer (with a tape-drive) and everything in P = NP is NP-complete, with two exceptions: the problem for which all instances are "yes" instances, and the problem for which all instances are

[‡]Maybe the magic box in *Sneakers* is a quantum computer!

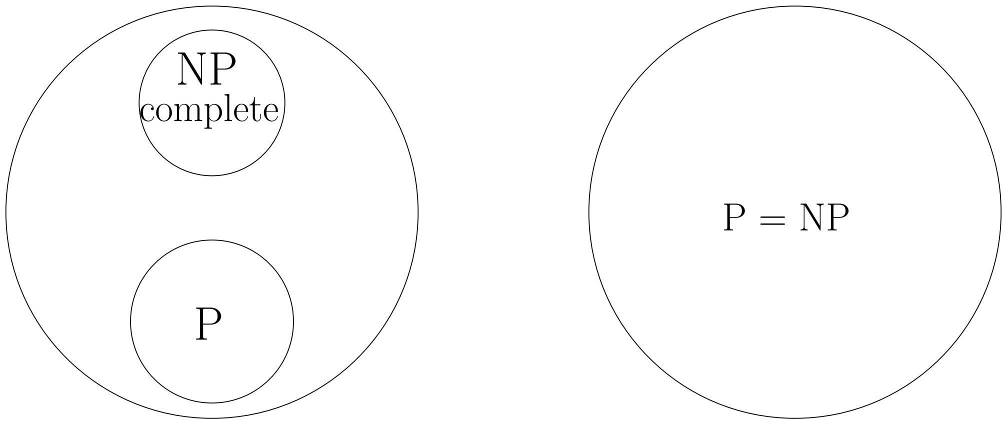


Figure 14.2: If P is a proper subset of NP, then NP looks like the diagram on the left: P and the class of NP-complete problems are disjoint. If P = NP, then NP looks like the diagram on the right: everything in P is NP-complete (with the exceptions of the problem for which all instances are "yes" instances and the problem for which all instances are "no" instances).

"no" instances (since we can't map a "no" instance of another problem to a "no" instance of the former, nor map a "yes" instance of another problem to a "yes" instance of the latter).

These two possibilities, P $\neq$ NP and P = NP, are roughly illustrated in Figure 14.2. If you want more details, I encourage you to ask a complexity theorist, who may get carried away and show you something like the map of complexity classes in Figure 14.3.

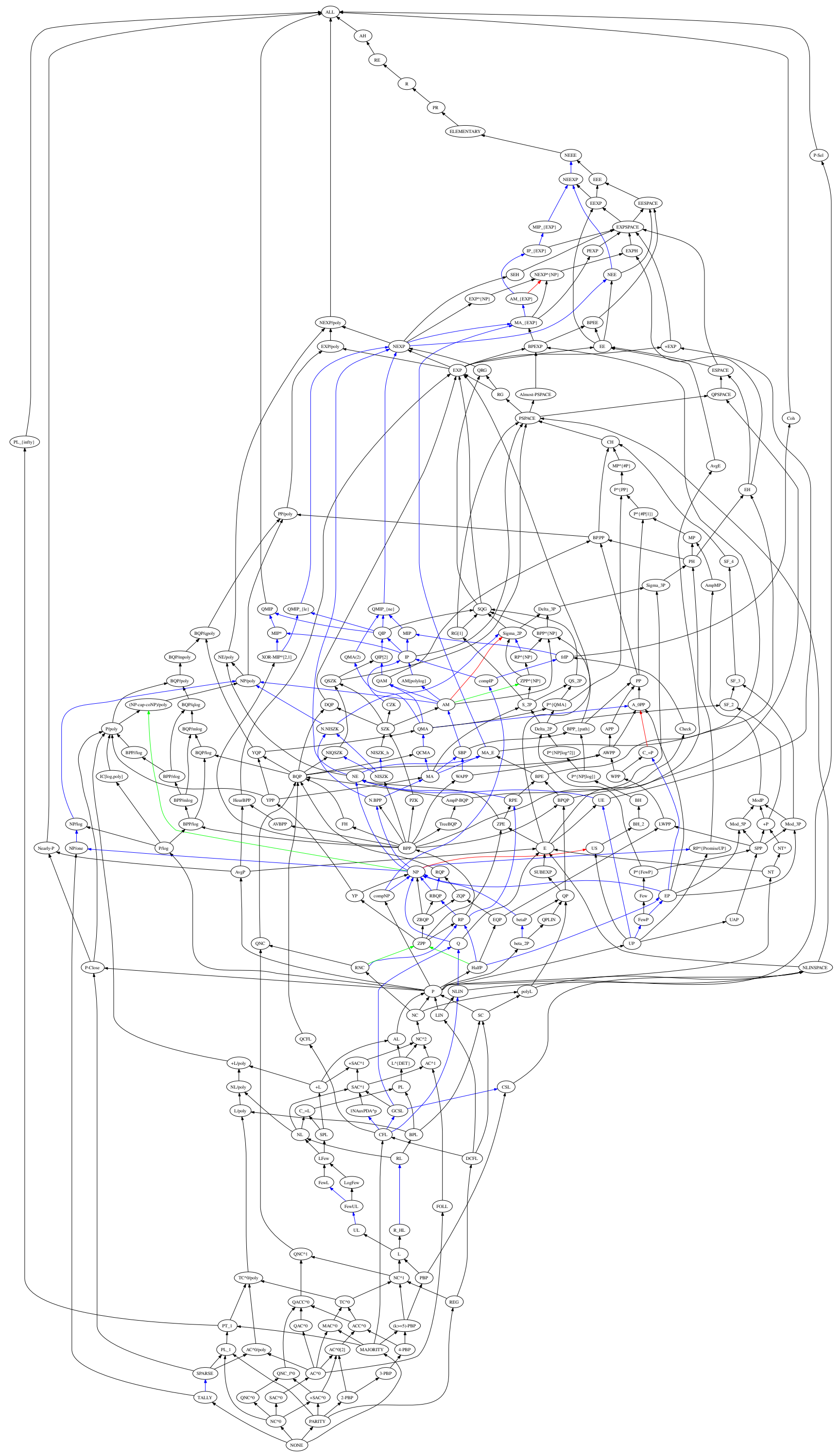


Figure 14.3: A map of complexity classes, from `https://www.math.ucdavis.edu/~greg/zoology`.

Assignment 6

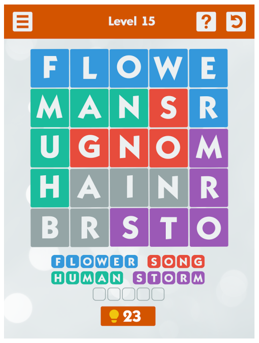
posted: 02.07.2021 **due: midnight 16.07.2021**

You can work in groups of up to three people. One group member should submit a copy of the solutions on Brightspace, with all members' names and banner numbers on it; the other group members should submit text files with all members' names and banner numbers (otherwise Brightspace won't let us assign them marks!). You may consult with other people but each group should understand the solutions: after discussions with people outside the groups, discard any notes and do something unrelated for an hour before writing up your solutions; it's a problem if no one in a group can explain one of their answers. For programming questions you should submit your code, which should compile and run correctly to receive full marks.

1. Show that Subset Sum is *self-reducible*: that is, if you have an algorithm that, given a set S of integers and a target t, in polynomial time (in the cardinality of S) determines whether a subset of S sums to t, then you can use it to design an algorithm that, given S and t, in polynomial time (in the cardinality of S) returns such a subset.

2. A *grid graph* is a graph whose vertices are labelled with distinct pairs of integers such that a vertex u labelled (x, y) is adjacent to a vertex v if and only if v is labelled $(x-1, y)$, $(x+1, y)$, $(x, y-1)$ or $(x, y+1)$. HamPath is known to be NP-complete even when restricted to grid graphs.
   For the problem Wordz 2, we are given an $n \times n$ grid of characters, a dictionary of strings and an integer k and asked if the grid contains at least k sequences of characters such that
   - if a character in a sequence has coordinates (x, y), then the next character must have coordinates $(x-1, y)$, $(x+1, y)$, $(x, y-1)$ or $(x, y+1)$,
   - each character in the grid appears at most once across all of the sequences,
   - each sequence is a distinct string in the dictionary.

   The image below shows an instance of Wordz 2 with a 5×5 grid and the dictionary of English words, with 4 sequences indicated with blue, red, green and purple. (The grey characters are not a sequence, although they could be.)
   Give a polynomial-time reduction from HamPath on Grid Graphs to Wordz 2. (You can assume all the coordinates in the vertices' labels in the graph are polynomial in the number of vertices.)



Continued on next page!

3. For the problem Integer Linear Programming (ILP) we are given a set of linear constraints such as

$$\begin{aligned} 3x_1 + 5x_2 &\leq 5 \\ 4x_2 - 2x_3 &= 10 \\ 6x_1 - x_2 + 3x_3 &\geq 6 \end{aligned}$$

and asked if there is a solution with all of the variables' values integers. Give a polynomial-time reduction from 3-Sat to ILP.

4. We can reduce 3-Col to Planar 3-Col in polynomial time. Why doesn't our $3^{O(n^{1/2} \log n)}$-time divide-and-conquer algorithm for Planar 3-Col give us a $3^{O(n^{1/2} \log n)}$-time algorithm for 3-Col?

5. We saw a 2-approximation algorithm for the search version of Vertex Cover, and the complement of a vertex cover is an independent set; does that mean we have a 2-approximation algorithm for the search version of Independent Set? Why or why not?

Midterm 2

posted: 9 am, 05.07.2021 **due: midnight, 09.07.2021**

You are **not** allowed to work in groups for the midterm. You can look in books and online, but you must not discuss the exam problems with anyone. If you don't understand a question, contact Travis or one of the TAs for an explanation (but no hints). All problems are weighted the same and, for problems broken down into subproblems, all subproblems are weighted the same.

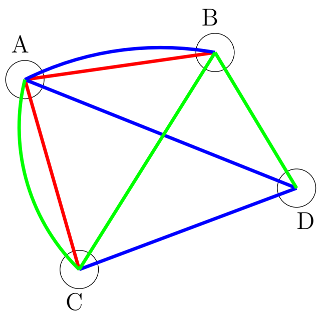
As with assignments, answers the markers consider hard to read will get 0!

1. Suppose your class is on a field trip to an island with n towns on it, connected by m town-to-town buses (which run in both directions), run by c companies. Each company's buses are a different colour, and there can be buses from two or more companies running between two towns. You have a map showing which companies run buses between which towns. The drivers have a relaxed attitude to schedules and the buses run often, so there's no telling which buses will be arriving and leaving next.

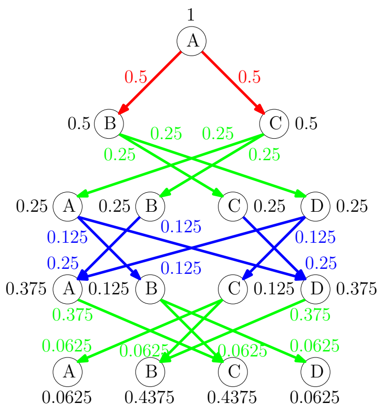
   Your classmate Ryan has wandered off and got lost and you're (somewhat reluctantly) trying to find him. You'd told him which buses the class was supposed to take during the day, and given him tickets from the appropriate companies, the same colours as the buses and stapled together in the right order. Ryan didn't remember which towns the class was going to visit, however, so he always took the first bus he saw of the colour of the next ticket, tearing off that ticket and giving it to the driver.

   Design a polynomial-time dynamic-programming algorithm that, given the map of the bus routes, Ryan's starting point and the colours of the buses he took, calculates the probability Ryan is in each of the n towns.

   For example, if the map is as shown below on the left and Ryan started in town A and took a red bus, a green bus, a blue bus and another green bus, then his itinerary could be any of those shown below in the center, and the probability of him being in a certain town after a certain number of steps and of taking trips between cities is as shown in the DAG below on the right.



A-B-C-D-B,
A-B-D-A-C,
A-B-D-C-A,
A-B-D-C-B,
A-C-A-B-C,
A-C-A-B-D,
A-C-A-D-B,
A-C-B-A-C .



The probability Ryan went first from A to B is 0.5, and the probability he went first from A to C is 0.5. Therefore, after one trip, the probability he was in B is 0.5 and the probability he was in C is 0.5.

The probability Ryan's second trip took him from B to C is 0.5 times the probability he was in B, or 0.25. The probability it took him from B to D is also 0.5 times the probability he was in B, or 0.25. The probability it took him from C to A is 0.5 times the probability he was in C, or 0.25. The probability it took him from C to B is 0.5 times the probability he was in C, or 0.25. Therefore, after two trips, the probability is 0.25 he was in any particular town.

The probability Ryan's third trip took him from A to B is 0.5 times the probability he was in A, or 0.125. The probability it took him from A to D is 0.5 times the probability he was in A, or 0.125. The probability it took him from B to A is the probability he was in A, or 0.25. The probability it took him from C to D is the probability he was in C, or 0.25. The probability it took him from D to A is 0.5 times the probability he was in D, or 0.125. The probability it took him from D to C is 0.5 times the probability he was in D, or 0.125.

Therefore, after three trips, the probabilities Ryan was in A, B, C, D are, respectively, $0.25 + 0.125 = 0.375$ (the probability his third trip took him from B to A plus the probability it took him from D to A), 0.125, 0.125 and $0.125 + 0.25 = 0.375$ (the probability his third trip took him from A to D plus the probability it took him from C to D).

You can compute the probability of Ryan being in a particular town after four trips similarly. Isn't it lucky you're on an island, so you can't accidentally lose Ryan forever?

(Hint: first design an algorithm that computes the number of ways Ryan could have ended up in a town, and then modify it to compute the probability.)

2. Your professor Travis told your TA Sarah that he was going to ask you to modify the solution to the alignment question on Assignment 5, to compute an optimal alignment using only one pass through the matrix.[§] In contrast, that assignment question allows filling in the matrix and then walking back from the bottom right corner to the top left corner to compute the alignment.

   Travis claimed it was possible by keeping two more arrays, top$[0..m, 0..n]$ and bottom$[0..m, 0..n]$, where top$[i, j]$ is a pointer to a string of length at most $m+n+1$ (including the end-of-string delimiter) containing the top line in an optimal alignment of $S[1..i]$ to $T[1..j]$, and bottom$[i, j]$ is a pointer to a string of length at most $m+n+1$ containing the bottom line in that alignment.

   To compute top$[i, j]$, we `sprintf` into an empty string either top$[i-1, j]$ or top$[i-1, j-1]$ or top$[i, j-1]$, followed by either $S[i-1]$ or '`-`'. To compute bottom$[i, j]$, we `sprintf` into an empty string either bottom$[i-1, j]$ or bottom$[i-1, j-1]$ or bottom$[i, j-1]$, followed by either $T[j-1]$ or `-`.

   Sarah correctly pointed out that `malloc`ing and `sprintf`ing a string of length $\Omega(m+n+1)$ takes $\Omega(m+n+1)$ time, so Travis's solution takes cubic time. Help Travis by figuring out how to modify his solution (shown below) to use quadratic time again.

   (Hint: pointers are your friends!)

[§]Yes, this question is based on a true story.

```
#include <stdio.h>
#include <stdlib.h>

#define MIN(a, b) (a < b ? a : b)
#define MIN3(a, b, c) (MIN(a, b) < c ? MIN(a, b) : c)

void align(char *S, char *T, int m, int n) {
  int A[m + 1][n + 1];
  char *top[m + 1][n + 1];
  char *bottom[m + 1][n + 1];

  A[0][0] = 0;
  top[0][0] = (char *) malloc(m + n + 1);
  bottom[0][0] = (char *) malloc(m + n + 1);
  sprintf(top[0][0], "");
  sprintf(bottom[0][0], "");

  for (int i = 1; i <= m; i++) {
    A[i][0] = i;
    top[i][0] = (char *) malloc(m + n + 1);
    bottom[i][0] = (char *) malloc(m + n + 1);
    sprintf(top[i][0], "%s%c", top[i - 1][0], S[i - 1]);
    sprintf(bottom[i][0], "%s-", bottom[i - 1][0]);
  }

  for (int j = 1; j <= n; j++) {
    A[0][j] = j;
    top[0][j] = (char *) malloc(m + n + 1);
    bottom[0][j] = (char *) malloc(m + n + 1);
    sprintf(top[0][j], "%s-", top[0][j - 1]);
    sprintf(bottom[0][j], "%s%c", bottom[0][j - 1], T[j - 1]);
  }

  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      A[i][j] = MIN3(A[i - 1][j] + 1, A[i][j - 1] + 1,
        A[i - 1][j - 1] + (S[i - 1] == T[j - 1] ? 0 : 1));

      top[i][j] = (char *) malloc(m + n + 1);
      bottom[i][j] = (char *) malloc(m + n + 1);

      if (A[i][j] == A[i - 1][j] + 1) {
        sprintf(top[i][j], "%s%c", top[i - 1][j], S[i - 1]);
        sprintf(bottom[i][j], "%s-", bottom[i - 1][j]);
      } else if (A[i][j] == A[i][j - 1] + 1) {
        sprintf(top[i][j], "%s-", top[i][j - 1]);
        sprintf(bottom[i][j], "%s%c", bottom[i][j - 1], T[j - 1]);
      } else {
        sprintf(top[i][j], "%s%c", top[i - 1][j - 1], S[i - 1]);
        sprintf(bottom[i][j], "%s%c", bottom[i - 1][j - 1], T[j - 1]);
      }
    }
  }

  printf("%s\n%s\n", top[m][n], bottom[m][n]);

  return;
}

int main() {
  char *S = "AGATACATCA";
  char *T = "GATTAGATACAT";

  align(S, T, 10, 12);

  return(0);
}
```

Bonus (worth 10% of the midterm): Can you reduce the space usage to $O((m+n)k)$, assuming you're given the edit distance k between S and T?

(Hint: banded dynamic programming and garbage collections are your friends!)

3. For the problem Longest Kind-Of Increasing Subsequence (LKOIS), we're given a sequence $S[1..n]$ of integers and asked to find the longest subsequence S' of S such that $S'[i-1] - 3 \leq S'[i]$ for $1 < i \leq |S'|$. Give an $O(n \log n)$ algorithm for LKOIS.

4. For the problem Partition, we're given a set S of positive integers that sum to $2t$ and asked if there is a subset of S that sums to exactly t. Prove Partition is NP-complete by
   - showing Partition is in NP,
   - reducing one of the NP-complete problems we've seen in class to Partition.

5. Write a program that, given a list of the edges in a connected graph G on the vertices $1, \ldots, n$, in polynomial time outputs a Boolean formula F that is satisfiable if and only if G has a Hamiltonian path. You can assume the list of edges looks something like

```
(1, 2)
(1, 3)
(4, 2)
(6, 5)
(5, 3)
```

   with one pair per line, and your output should consist of a single line containing copies of space, `(`, `)`, `AND`, `OR`, `NOT` and variables that look something like `x1`, `x2`, etc.

Part V

Last Assignments and Final

Assignment 7

posted: 16.07.2021 **due: midnight 23.07.2021**

You can work in groups of up to three people. One group member should submit a copy of the solutions on Brightspace, with all members' names and banner numbers on it; the other group members should submit text files with all members' names and banner numbers (otherwise Brightspace won't let us assign them marks!). You may consult with other people but each group should understand the solutions: after discussions with people outside the groups, discard any notes and do something unrelated for an hour before writing up your solutions; it's a problem if no one in a group can explain one of their answers. For programming questions you should submit your code, which should compile and run correctly to receive full marks.

1. You may have seen the standard BFS-based algorithm that, given a set of n jugs with capacities $c_1, \ldots, c_n$ in litres, a target of t litres and access to a tap, finds the fastest way of using the jugs to measure out t litres of water, or that it's not possible. If the c_is are integers then this takes $O((c_1+1) \cdot (c_2+1) \cdots (c_n+1) \cdot (n+1)^2)$ time.

   How can you modify the standard algorithm such that it runs in
   $$O((c_1+1) \cdot (c_2+1) \cdots (c_n+1) \cdot (n+1)^2 \cdot \log((c_1+1) \cdot (c_2+1) \cdots (c_n+1))$$
   time and, instead of the fastest way (fewest pours), it finds the way to measure out t litres which involves the least total lifting? For example, pouring 2 litres from 5-litre jug containing 4 litres into a 3-litre jug containing 1 litre, and then pouring the remaining 2 litres from the 5-litre jug into a 6-litre jug containing 1 litre, involves lifting a jug containing 4 litres and then lifting a jug (the same one) that contains 2 litres, so the total cost is 6. (You can assume the tap has a hose so you can fill a jug without lifting it.)

2. Suppose you're given a list of statements such as "Factoring polytime reduces to Sat", "Clique polytime reduces to Independent Set", "Independent Set polytime reduces to Sat" and "Sat polytime reduces to Clique". How can you divide the mentioned problems up into the minimum number of equivalence classes such that, for any equivalence class C and any two problems P and Q in C, you know only from the statements that P polytime reduces to Q and vice versa? (In the example above, there are two equivalence classes: {Factoring} and {Sat, Clique, Independent Set}.)

3. How can you modify Dijkstra's (without making it much slower) such that it works even when there's *one* directed negative-weight edge in the graph?

4. How can you determine if there's a negative-weight cycle of length at most k in time $O(kn^3)$, where n is the number of vertices in the graph?

5. Give an $O(n^3 \log k)$-time algorithm that, given an integer k and the $n \times n$ adjacency matrix of a graph on n vertices, returns the $n \times n$ matrix in which cell (i, j) is the number of ways of going from the ith vertex to the jth vertex in exactly k steps.

   (Hint: First figure out how to do this when k is a power of 2, and then consider the binary representation of general k.)

Assignment 8

posted: 23.07.2021 **due: midnight 29.07.2021**

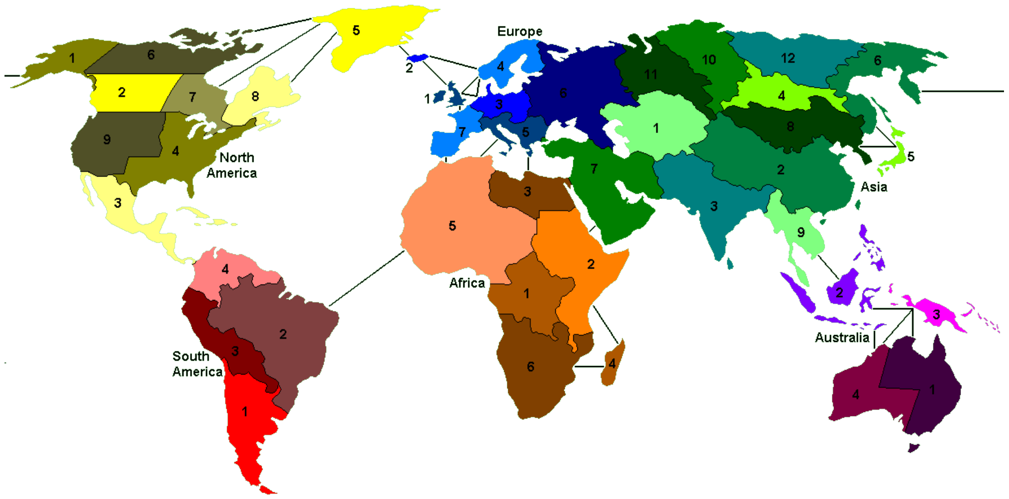
You can work in groups of up to three people. One group member should submit a copy of the solutions on Brightspace, with all members' names and banner numbers on it; the other group members should submit text files with all members' names and banner numbers (otherwise Brightspace won't let us assign them marks!). You may consult with other people but each group should understand the solutions: after discussions with people outside the groups, discard any notes and do something unrelated for an hour before writing up your solutions; it's a problem if no one in a group can explain one of their answers. For programming questions you should submit your code, which should compile and run correctly to receive full marks.

1. Consider the map from Assignment 1, shown below. Suppose you and lots of your friends have decided to celebrate the end (?) of the pandemic with trips from Dal to Eastern Europe. To maintain the feeling that the world is big and wide, you don't want to travel in big groups, nor keep running into each other while you're travelling.

   You've decided to stay spread out with the following rule: on any particular day, if region A is labelled with the number x and region B is labelled with the number y, then at most $\min(x, y)$ people can travel from A to B, and at most $\min(x, y)$ can travel from B to A. (You can assume movements are synchronous.)

   The first few days may be a little chaotic, but then things should stabilize for a while, with the same number of people leaving each region as entering, until the last people are leaving Dal and eventually making their way to Eastern Europe. During this intermediate period of stability, how many people are leaving Dal each day (or, equivalently, how many are arriving in Eastern Europe)?

   (Hint: can you find a matching cut?)



2. Suppose you're organizing a dinner at an event with n students from k universities and trying to choose a seating plan. Eight people can sit at each table and you don't want more than

three people from the same university sitting at the same table. How can you efficiently find the minimum number of tables you'll need? The input is the number of people from each university, $n_1, \dots, n_k$ with $n_1 + \cdots + n_k = n$, and the output is the number of tables.

(Hint: to test if t tables are enough, create a graph G_t with a source, a sink, a vertex for each university, and a vertex for each of t tables.)

3. We saw in the Lecture 19 that if we assume there's a C routine P that, given any string S, returns the length in characters of the shortest C program that outputs S and then stops, then we reach a contradiction.

   Specifically, if the code for P looks like

```
int P (char *S) {
  ...SOME CODE GOES HERE...
}
```

   (with the code to compute P replacing `...SOME CODE GOES HERE...`), then we can write a program Q that has P as a subroutine and loops through all possible strings in increasing order by length until it finds one that P says requires a program much longer than Q, at which point Q stops and outputs that string. (P must eventually say some string requires a program much longer than Q, by counting arguments.) An example of such a program Q is shown below.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct node {
  char *string;
  struct node *next;
} node;

char *stringP = "int P (char *S) {\n  ...SOME CODE GOES HERE...\n}";
int lenP = strlen(stringP);

int P (char *S) {
  ...SOME CODE GOES HERE...
}

int main() {
  node *head = (node *) malloc(sizeof(node));
  node *tail = head;

  tail -> string = (char *) malloc(1);
  sprintf(tail -> string, "");
  tail -> next = NULL;
  while (1) {
    if (P(head -> string) > 2 * lenP + 1000000) {
    // Notice P appears twice in this program:
    // once as a string and once as a subroutine.
    // The number 1000000 only has to be more than
    // the length of the rest of the program.
      printf("%s", head -> string);
      return(0);
    } else {
      for (int c = 0; c < 256; c++) {
        tail -> next = (node *) malloc(sizeof(node));
        tail = tail -> next;
        tail -> string = (char *) malloc(strlen(head -> string) + 2);
        sprintf(tail -> string, "%s%c", head -> string, (char) c);
        tail -> next = NULL;
      }
      head = head -> next;
    }
  }
}
```

Adapt that argument to show it's not possible even to *approximate within a factor of 10* the length of the shortest C program that outputs S and then stops. How does the code for Q above change?

Final Exam

posted: 8 am, 05.08.2021 **due: midnight, 05.08.2021**

You are **not** allowed to work in groups for the final exam. You can look in the lecture notes, books and online, but you must not discuss the exam problems with anyone. If you don't understand a question, contact Travis or one of the TAs for an explanation (but no hints). All problems are weighted the same and, for problems broken down into subproblems, all subproblems are weighted the same. **Answers that are difficult to read will receive 0.**

1. Suppose you are given a rooted tree T on n vertices, each of which is assigned a colour. (In this problem, both endpoints of an edge can have the same colour.) Give an efficient divide-and-conquer algorithm to find the longest path in T whose vertices are all the same colour. (Paths can ascend and then descend, as long as they don't revisit vertices.) What is the complexity of your algorithm? **You need not prove your algorithm correct.**

2. (a) Give an efficient greedy algorithm that, given a sequence $A[1..n]$ of integers, partitions A into the minimum number of kind-of increasing subsequences. A subsequence is *kind-of increasing* if, for each consecutive pair of numbers $A[h]$ and $A[i]$ in the subsequence, with $h < i$, we have $A[h] - 3 \leq A[i]$. **Prove your algorithm correct.**

   (b) What goes wrong if you try to partition S into the number of slowly increasing subsequences the same way? A subsequence is *slowly increasing* if, for each consecutive pair of numbers $S[i]$ and $S[j]$ in the subsequence, with $i < j$, $S[i] < S[j] \leq S[i] + 10$?

   (Hint: consider the two sequences 8, 1, 9, 2 and 8, 1, 9, 18, 18.)

3. Suppose you are given a directed acyclic graph G on n vertices, each of which is assigned a colour. (In this problem, both endpoints of an edge can have the same colour.) Give an efficient dynamic-programming algorithm to find the length of the longest directed path in G whose vertices are all the same colour. What is the complexity of your algorithm? **You need not prove your algorithm correct.**

   (Hint: start with a topological sort.)

4. Suppose you are given a graph G on n vertices, each of which is assigned a colour. (In this problem, both endpoints of an edge can have the same colour.) Either give an efficient algorithm to find the longest monochromatic path in G or justify your inability to do so. **If you give an algorithm, you need not prove it correct.**

5. We reduced 3-Col to Planar 3-Col in polynomial time using a *crossing gadget*. Why can't we reduce 4-Col to Planar 4-Col in polynomial time the same way (assuming P $\neq$ NP)? What goes wrong in the reduction?

6. Suppose you are given a *directed* graph G on n vertices, each of which is assigned a colour. (In this problem, both endpoints of an edge can have the same colour.) Give an efficient algorithm to find the length of the longest monochromatic directed walk in G. (In a walk we can revisit vertices and recross edges, whereas in a path we cannot.) Your algorithm should return "undefined" if there is a directed cycle whose vertices are all the same colour.

Part VI

Solutions

Assignment 1 Solution

The solution below is not pretty — as one of you amusingly wrote in the Zoom chat when I showed you my code for counting the 3-colourings of Nova Scotia, “my eyes are on fire, it’s so ugly” — but as I said, I’m not going to give out a general solution because I want to be able to reuse this exercise if Dal has me teach this course again, just changing the map. The code I originally posted had a couple of bugs in it and the answer it returned was about 3 times larger than what the code below returns (and what we now think is the real answer), 8720115499008. Both versions ran in a second or two on my very standard, two-year-old laptop. As long as your answer was within a factor of 1000 either way and wasn’t obviously wrong, we’d consider it correct.

```
#include <stdio.h>

long long int countA();
long long int countB();
long long int countC();
long long int countD();
long long int countE();
long long int countF();

long long int A1, A2, A3, A4, A5, A6;
long long int B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11, B12;
long long int C1, C2, C3, C4;
long long int D1, D2, D3, D4, D5, D6, D7;
long long int E1, E2, E3, E4, E5, E6, E7, E8, E9;
long long int F1, F2, F3, F4;

int main() {
long long int counter = 0, temp;

 for (A5 = 0; A5 < 4; A5++) {
  for (B6 = 0; B6 < 4; B6++) {
   for (B7 = 0; B7 < 4; B7++) {
    for (B9 = 0; B9 < 4; B9++) {
     for (D5 = 0; D5 < 4; D5++) {
      if (D5 != A5 && D5 != B7) {
       for (D6 = 0; D6 < 4; D6++) {
        if (D6 != B7 && D6 != D5) {
         for (E3 = 0; E3 < 4; E3++) {
          for (E5 = 0; E5 < 4; E5++) {
           counter += countA() * countB() * countC() * countD() * countE() * countF();
 }}}}}}}}}}
 printf("The number of 4-colourings is %lli.\n", counter);
 return(0);
}

long long int countA() {
 long long int counterA = 0;

 for (A1 = 0; A1 < 4; A1++) {
  if (A1 != A5) {
   for (A2 = 0; A2 < 4; A2++) {
    if (A2 != A1 && A2 != A5 && A2 != B7) {
     for (A3 = 0; A3 < 4; A3++) {
      if (A3 != A2 && A3 != A5 && A3 != B7 && A3 != D5) {
       for (A4 = 0; A4 < 4; A4++) {
        if (A4 != A2) {
         for (A6 = 0; A6 < 4; A6++) {
          if (A6 != A1 && A6 != A2 && A6 != A4) {
           counterA++;
 }}}}}}}}}}
 return(counterA);
}

long long int countB() {
 long long int counterB = 0;

 for (B1 = 0; B1 < 4; B1++) {
  if (B1 != B7 && B1 != D6) {
   for (B2 = 0; B2 < 4; B2++) {
    if (B2 != B1 && B2 != B9) {
     for (B3 = 0; B3 < 4; B3++) {
      if (B3 != B1 && B3 != B2 && B3 != B7 && B3 != B9) {
       for (B4 = 0; B4 < 4; B4++) {
        if (B4 != B6) {
         for (B5 = 0; B5 < 4; B5++) {
          if (B5 != B6) {
           for (B8 = 0; B8 < 4; B8++) {
            if (B8 != B2 && B8 != B4 && B8 != B5 && B8 != B6) {
             for (B10 = 0; B10 < 4; B10 ++) {
              if (B10 != B2 && B10 != B4 && B10 != B8) {
               for (B11 = 0; B11 < 4; B11++) {
                if (B11 != B1 && B11 != B2 && B11 != B10 && B11 != D6) {
                 for (B12 = 0; B12 < 4; B12++) {
                  if (B12 != B4 && B12 != B6 && B12 != B10) {
                   counterB++;
 }}}}}}}}}}}}}}}}}}
 return(counterB);
}

long long int countC() {
 long long int counterC = 0;

 for (C1 = 0; C1 < 4; C1++) {
  for (C2 = 0; C2 < 4; C2++) {
   if (C2 != B9) {
    for (C3 = 0; C3 < 4; C3++) {
     if (C3 != C1 && C3 != C2) {
      for (C4 = 0; C4 < 4; C4++) {
       if (C4 != C1 && C4 != C2 && C4 != C3) {
        counterC++;
 }}}}}}}
 return(counterC);
}

long long int countD() {
 long long int counterD = 0;

 for (D1 = 0; D1 < 4; D1++) {
  for (D2 = 0; D2 < 4; D2++) {
   if (D2 != D1 && D2 != E5) {
    for (D3 = 0; D3 < 4; D3++) {
     if (D3 != D1 && D3 != D5 && D3 != D6) {
      for (D4 = 0; D4 < 4; D4++) {
       if (D4 != D1 && D4 != D2 && D4 != D3 && D4 != D6) {
        for (D7 = 0; D7 < 4; D7++) {
         if (D7 != A5 && D7 != D1 && D7 != D3 && D7 != D5) {
          counterD++;
 }}}}}}}}}
 return(counterD);
}

long long int countE() {
 long long int counterE = 0;

 for (E1 = 0; E1 < 4; E1++) {
  if (E1 != B6) {
   for (E2 = 0; E2 < 4; E2++) {
    if (E2 != E1) {
     for (E4 = 0; E4 < 4; E4++) {
      if (E4 != E3) {
       for (E6 = 0; E6 < 4; E6++) {
        if (E6 != E1 && E6 != E2 && E6 != E5) {
         for (E7 = 0; E7 < 4; E7++) {
          if (E7 != E2 && E7 != E4 && E7 != E5 && E7 != E6) {
           for (E8 = 0; E8 < 4; E8++) {
            if (E8 != E4 && E8 != E5 && E8 != E7) {
             for (E9 = 0; E9 < 4; E9++) {
              if (E9 != E2 && E9 != E3 && E9 != E4 && E9 != E7) {
               counterE++;
 }}}}}}}}}}}}}
 return(counterE);
}

long long int countF() {
 long long int counterF = 0;

 for (F1 = 0; F1 < 4; F1++) {
  for (F2 = 0; F2 < 4; F2++) {
   if (F2 != A5 && F2 != F1) {
    for (F3 = 0; F3 < 4; F3++) {
     if (F3 != F1 && F3 != F2) {
      for (F4 = 0; F4 < 4; F4++) {
       if (F4 != E3 && F4 != F2 && F4 != F3) {
        counterF++;
 }}}}}}}
 return(counterF);
}
```

Assignment 2 Solutions

1. Given a tree on n vertices, we can always find a single vertex whose removal leaves a forest in which no tree has more than $n/2$ vertices. Suppose we use our divide-and-conquer algorithm to count the 3-colourings of a tree on n vertices; about how long does it take? How fast can you compute the answer?
   You can assume n is a power of 2 and the tree is always split into exactly two pieces of size $n/2$ (even though the two pieces together should have $n-1$ vertices instead of n, since we removed a vertex to split the tree).

   Solution: In this case, the recurrence for our colouring algorithm from Assignment 1 is

$$\begin{aligned}T(n) &= 3 \cdot 2T(n/2) + f(n)\\ &= 6(6T(n/4) + f(n/2)) + f(n)\\ &= 6(6(6T(n/8) + f(n/4)) + f(n/2)) + f(n)\\ &= 6^i T(n/2^i) + \sum_{j=0}^{i-1} 6^j f(n/2^j)\,.\end{aligned}$$

   I forgot to say it takes $O(n)$ time to find the vertex whose removal leaves a forest in which no tree has more than $n/2$ vertices — that is, $f(n) = n$ — but trying every vertex naïvely would take $O(n^2)$ time. Plugging that in and setting $i = \lg n$, we still get

$$T(n) \approx 6^{\lg n} + n^2 \sum_{j=0}^{\lg n - 1} (3/2)^j = O(6^{\lg n}) = O(n^{\lg 6})\,.$$

   On the other hand, it doesn't actually matter what the shape of the tree is: for whichever vertex you decide to colour first, you have 3 choices; for all its neighbours, you have 2 choices; for all their neighbours, you have 2 choices; etc. So the number of ways to 3-colour a tree on n vertices is always $3 \cdot 2^{n-1}$. (That's a big number, but it's easy to work with because in binary it's just 11 followed by $n-1$ copies of 0.)

2. In the lecture, we saw that implementing Euclid's algorithm on positive integers a and b with $a > b$ by repeated subtraction takes $\Omega(a)$ time in the worst case but implementing it by mod takes $O(\log a)$ time, *assuming subtraction and* mod *each take constant time.* Now suppose subtracting two n-digit numbers takes n time but taking their mod takes n^2 time; comparing two numbers takes time 1. About how much bigger does a have to be than b in order for it to be faster to compute $a \bmod b$ with mod directly than with repeated subtraction?
   For example, if $a = 1523$ and $b = 0427$, then computing $a \bmod b = 242$ by repeated subtraction means subtracting 0427 from 1523 to get 1096 in 4 time units, checking 1096 is still bigger than 0427 in 1 time unit, subtracting 0427 from 1096 to get 0669 in 4 time units, checking 0669 is still bigger than 0427 in 1 time unit, subtracting 0427 from 0667 to get 0242 in 4 time units, and checking whether 0240 is bigger than 0427 in 1 time unit (and finding it's not). That takes a total of $4 + 1 + 4 + 1 + 4 + 1 = 15$ time units, whereas computing $a \bmod b = 242$ directly takes $4^2 = 16$ time units, so in this case repeated subtraction is faster.

Solution: It's faster to compute $a \bmod b$ when a is at least about n times larger than b: if $a/b = \omega(n)$ then it will take $\omega(n^2)$ time to do the repeated subtractions; if $a/b = \Theta(n)$ then it will take $\Theta(n^2)$ time to do either the repeated subtractions or compute the modulus; if $a/b = o(n)$ then it will take $o(n^2)$ time to do the repeated subtractions.

3. Describe how to build a circuit consisting of AND, OR and NOT gates that takes two n-bit binary numbers x and y and outputs the $(n+1)$-bit binary number $x+y$. Your circuit should be a directed acyclic graph (a DAG) whose size is at most polynomial in n and whose depth is constant (where "depth" means the length of the longest directed path); the fan-in and fan-out are not bounded (where "fan-in" and "fan-out" mean the maximum in- and out-degree of any vertex).

   Solution: To be consistent with what I wrote in the discussion on Brightspace, let's number the columns from the right, say the sum of x and y is z, and consider how to compute the bit $z[i]$.
   If we know the carry-bit $c[i]$ from adding $x[i-1..0]$ and $y[i-1..0]$, then for $1 \le i < n$ we can compute $z[i]$ according to the following truth table:

$x[i]$	$y[i]$	$c[i]$	$z[i]$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

   That is, $z[i]$ is 1 if and only if an odd number of $x[i]$, $y[i]$ and $c[u = i]$ are 1s, for $1 \le i < n$. (If $i = 0$ then we ignore the carry-bit, and $z[n] = c[n]$.)
   As people pointed out in the discussion, that still leaves the problem of figuring out the carry-bit $c[i]$. Notice that $c[i] = 1$ if and only if, for some $k < i$, we have $x[k] = y[k] = 1$ and, for all j with $i > j > k$, we do *not* have $x[j] = y[j] = 0$. In other words (or, rather, symbols),

   $$c[i] = \bigvee_{k<i} \left(x[k] \wedge y[k] \wedge \left(\bigwedge_{i>j>k} (x[j] \vee y[j]) \right) \right) .$$

   For each choice of i and $k < i$ and j with $i > j > k$, we have an AND gate, so our entire circuit has $\Theta(n^3)$ gates — but that's allowed. The main thing is, the entire circuit has constant depth.

4. Give a divide-and-conquer program for

   `https://leetcode.com/problems/maximum-subarray`

   (you don't have to pay for a membership!) and explain how to use your solution to solve

   `https://leetcode.com/problems/maximum-sum-circular-subarray` neatly.

(If you don't use divide-and-conquer or your solution looks like it's been copied, you will *not* get the mark and you may be reported to FCS.)

Solution: The main idea is that the max-sum subarray `nums`$[i..j]$ is either entirely in the first half of `nums`, so $j \leq$ `numsSize`$/2$; or entirely in the second half, so $i >$ `numsSize`$/2$; or crosses the middle, in which case its a non-empty suffix of the first half followed by a non-empty prefix of the second half, so $i \leq$ `numsSize`$/2 < j$.
We find the max-sum subarrays in `nums`$[0..\lfloor$`numsSize`$/2\rfloor]$ and `nums`$[\lfloor$`numsSize`$/2\rfloor..$`numsSize`$-1]$ recursively. With a for-loop, we work out the sum of each non-empty suffix of `nums`$[0..\lfloor$`numsSize`$/2\rfloor]$ and, with another for-loop we work out the sum of each non-empty prefix of `nums`$[\lfloor$`numsSize`$/2\rfloor..$`numsSize`$-1]$. We concatenate the max-sum suffix of the first half and the max-sum prefix of the second half to get the max-sum subarray that crosses the middle. We return the largest of the sums of the max-sum subarray entirely in the first half, the max-sum subarray entirely in the second half, and the max-sum subarray that crosses the middle.

```
int maxSubArray(int* nums, int numsSize){

    if (numsSize == 1) {
        return(nums[0]);
    }

    int split = numsSize / 2;
    int maxLeft = maxSubArray(nums, split);
    int maxRight = maxSubArray(&nums[split], numsSize - split);

    int sufSum = nums[split - 1];
    int maxSufSum = sufSum;
    for (int i = split - 2; i >= 0; i--) {
        sufSum += nums[i];
        if (sufSum > maxSufSum) {
            maxSufSum = sufSum;
        }
    }

    int prefSum = nums[split];
    int maxPrefSum = prefSum;
    for (int i = split + 1; i < numsSize; i++) {
        prefSum += nums[i];
        if (prefSum > maxPrefSum) {
            maxPrefSum = prefSum;
        }
    }

    int maxMiddle = maxSufSum + maxPrefSum;
    if (maxLeft >= maxRight && maxLeft >= maxMiddle) {
        return(maxLeft);
    } else if (maxRight >= maxMiddle) {
        return(maxRight);
    } else {
        return(maxMiddle);
    }
}
```

The coolest way to reuse this code to find the max-sum subarray in a circular array A is to find the max-sum subarray $A[i..j]$ in A considering it as a normal array, and then find the max-sum subarray $A'[i'..j']$ in A' where $A'[k] = -A[k]$ for all k, and then report the larger of the sums of $A[i..j]$, and of $A[j'+1..|A|-1]$ concatenated with $A[0..i'-1]$.
The more obvious way to find the max-sum subarray is to find the max-sum subarray $A[i..j]$, find the max-sum prefix of A, find the max-sum suffix of A, and then report the larger of the sums of $A[i..j]$, and of the max-sum suffix concatenated with the max-sum prefix.

5. Suppose you didn't understand Max's lecture on the FFT but you still want to multiply degree-n polynomials in time subquadratic in n. Show how to use Karatsuba's algorithm to do it in $O(n^{\lg 3})$ time, assuming arithmetic operations on coefficients take constant time.
For example, consider multiplying the two polynomials

$$\begin{aligned} A(x) &= 8x^7 + 5x^6 + 4x^5 + x^4 + 9x^3 + 6x^2 + 2x + 1 \\ B(x) &= 7x^7 + 5x^6 + 3x^5 + 3x^4 + 9x^3 + 4x^2 + 5\,. \end{aligned}$$

Notice

$$\begin{aligned} &(8x^3 + 5x^2 + 4x + 1)(9x^3 + 4x^2 + 5) + (9x^3 + 6x^2 + 2x + 1)(7x^3 + 5x^2 + 3x + 3) \\ &\quad = (8x^3 + 5x^2 + 4x + 1 + 9x^3 + 6x^2 + 2x + 1)(7x^3 + 5x^2 + 3x + 3 + 9x^3 + 4x^2 + 5) - \\ &\qquad (8x^3 + 5x^2 + 4x + 1)(7x^3 + 5x^2 + 3x + 3) - (9x^3 + 6x^2 + 2x + 1)(9x^3 + 4x^2 + 5)\,; \end{aligned}$$

does this look familiar?

Solution: We can imagine x is a base so large that we'll never have to worry about carries, and then apply Karatsuba's algorithm.
In the example, we set

$$\begin{aligned} a_1(x) &= 8x^3 + 5x^2 + 4x + 1 \\ a_0(x) &= 9x^3 + 6x^2 + 2x + 1 \\ b_1(x) &= 7x^3 + 5x^2 + 3x + 3 \\ b_0(x) &= 9x^3 + 4x^2 + 0x + 5 \end{aligned}$$

so

$$\begin{aligned} A(x) &= a_1(x)x^4 + a_0(x) \\ B(x) &= b_1(x)x^4 + b_0(x)\,. \end{aligned}$$

and

$$\begin{aligned}
A(x)B(x) &= (a_1(x)x^4 + a_0(x))(b_1(x)x^4 + b_0(x)) \\
&= a_1(x)b_1(x)x^8 + (a_1(x)b_0(x) + a_0(x)b_1(x))x^4 + a_0(x)b_0(x)\,.
\end{aligned}$$

Setting

$$\begin{aligned}
z_2(x) &= a_1(x)b_1(x) \\
z_0(x) &= a_0(x)b_0(x) \\
z_1(x) &= a_1(x)b_0(x) + a_0(x)b_1(x) \\
&= (a_1(x) + a_0(x))(b_1(x) + b_0(x)) - z_2 - z_0
\end{aligned}$$

we get

$$A(x)B(x) = z_2(x)x^8 + z_1(x)x^4 + z_0(x)\,.$$

Assignment 3 Solutions

1. Mark the true statments. (Your score for this questions will be proportional to the number of statements you mark correctly, plus the number you correctly leave unmarked, minus the number you should have marked but didn't, minus the number you shouldn't have marked but did.)

   (a) $\lg n^n = O(\lg n!)$
   (b) $n^{(n+1) \bmod 2} = \Omega(n^{n \bmod 2})$ for $n \in \mathbb{N}$
   (c) $n^{(3n) \bmod 2} = O(n^{n \bmod 2})$ for $n \in \mathbb{N}$
   (d) If $T(n) = 7T(n/3) + n^4$ then $T(n) = \Omega(n^4/\log n)$.
   (e) If $T(n) = 8T(n/2) + 8n^3$ then $T(n) = \Theta(n^3 \log n)$.

   Solution: a, c, d, e

2. Explain how you can sort a sequence of n integers from a range of size $n^{\lg \lg n}$ in $O(n \log \log n)$ time (assuming each integer fits in a constant number of machine words).

   Solution: Consider each number as a $(\lg \lg n)$-digit number in base $b = 2^{\lceil \lg n \rceil}$. Since b is linear in n, we can radix-sort all the numbers in $O(n)$ time for each column of digits, so we use $O(n \log \log n)$ time overall.

3. An *in-place* algorithm uses a constant number of machine words of memory on top of the memory initially occupied by its input and eventually occupied by its output. Give code or pseudo-code for an in-place version of QuickSort.

   Solution: Start a pointer p at the start of the subarray and a pointer q at the end of the subarray; move p right until we find a value larger than the pivot and move q left until we find a value smaller than the pivot; then swap the elements pointed to by p and q; continue. We stop when p meets q.

4. You've probably seen in previous courses how to build a min-heap on n elements in $O(n)$ time and how to extract the minimum value from one in $O(\log n)$ time. Do you think we can easily extract the minimum value in $o(\log n)$ time while still leaving a heap on the remaining elements? Why or why not?

   Solution: If we could do this easily, then we could easily implement HeapSort in $o(n \log n)$ time.

5. Suppose you have an algorithm that, given a sequence of n integers that can be partitioned into d non-decreasing subsequences but not fewer, does so in $O(n \log d)$ time. Explain how you can also sort such a sequence in $O(n \log d)$ time.

   Solution: We partition the sequence into the d subsequences, and then merge them in $O(nlogd)$ time using a min-heap on d elements, containing the first element in each list. When we extract the minimum element, we insert into the heap the next element from the list that extracted element came from, maintaining the invariant that the smallest element from each list not yet included in the sorted output, is in the heap.

Assignment 4 Solutions

1. You have a week to complete an assignment with several questions, each worth the same number of marks. You don't want to spend more than h hours on the whole assignment and you can estimate accurately how many hours each question will take you. Give a greedy algorithm to decide which questions to answer. PROVE YOUR ALGORITHM CORRECT!

   Solution: Sort the questions into increasing order by the time it takes you to answer them and do as many you can in that order.

   > **Before we take any steps, our (empty) subsolution can be extended to an optimal solution. Assume that, after $i \geq 0$ steps, our subsolution can be extended to an optimal solution S. Then we show that after $i + 1$ steps, our subsolution can be extended to an optimal solution S'. Therefore, by induction, we obtain an optimal solution.**

   If we do not answer the $(i+1)$st question q, then we do not have enough time for it, so S does not answer it either and $S' = S$. If we answer q and so does S, then $S' = S$. If we answer the q and S does not, then whichever is the question q' that S answers instead, we can swap q and q' and obtain a solution S' that is optimal because it answers as many questions as S, and extends our subsolution after $i + 1$ steps. (If S doesn't answer any questions instead of q, then we can add q to S to obtain S'.)

2. Your professor is training to run against his friend Simon, but he's not sure he can make it around his whole planned route in one go, so he's made a list of places where he can stop for a break, coffee, etc. (For example, if he starts at the shipyards and runs along the coast, he can stop at the Tim Horton's by the ferry terminal, then in the Salt Yard, then at one of the restaurants along the waterfront, then at the Garrison Brewery or Tomavinos by the Seaport Market, then at the entrance of Point Pleasant, then at the top of Arm Road in the park, etc etc.) Suppose he gives you this list, with the distance between each consecutive pair of potential pit stops, and the distance d he can run without stopping. Give a greedy algorithm that tells him where to stop such that 1) he never runs more that distance d without a break and 2) he makes the minimum number of stops. PROVE YOUR ALGORITHM CORRECT!

   Solution: Considering his starting point as the first pit stop, after each pit stop p he should run to the furthest pit stop that is within distance d of p.

   > **Before we take any steps, our (empty) subsolution can be extended to an optimal solution. Assume that, after $i \geq 0$ steps, our subsolution can be extended to an optimal solution S. Then we show that after $i + 1$ steps, our subsolution can be extended to an optimal solution S'. Therefore, by induction, we obtain an optimal solution.**

   If you tell your professor to stop at pit stop $i + 1$ and so does S, then $S' = S$. If you tell him to keep going and so does S, then $S' = S$.
   Suppose you tell your professor to stop at pit stop $i + 1$ and S tells your professor to keep going. Since you always have your professor as far as possible without going over distance d, following S will cause him to collapse or something.

Now suppose S tells your professor to stop at pit stop $i+1$ and you tell him to keep going to pit stop j, with $j > i+1$. Since you never tell your professor to run further than distance d, he'll make it to pit stop j. Let k be the first pit stop after j that S tells your professor to stop at. Since S has your professor run from a pit stop before or equal to j (notice it could have him stop more times getting to j, even if that would be a waste) to pit stop k, the distance from j to k is at most d. Therefore, you can tell your professor that, after j, he should stop at the pit stops in S at k or later, and get a solution S' with at most as many stop as S that extends your solution after $i+1$ steps.
(A step here is a pit stop, whether you say to skip it or stop at it. This is basically problem 16.2-4 from *Introduction to Algorithms.*)

3. A *cross parsing* of a string $S[1..m]$ with respect to a string $T[1..n]$ is a partition of S into a minimum number of substrings each of which occurs in T. Suppose you are given an array $L[1..m]$ such that, for $1 \leq i \leq m$, the substring $S[i..i+L[i]-1]$ occurs in T but the substring $S[i..i+L[i]]$ doesn't. Give a greedy algorithm for computing a cross parsing of S with respect to T. PROVE YOUR ALGORITHM CORRECT!

   Solution: We make the first substring $S[1..L[1]]$. If a substring ends at $S[j-1]$, then we make the next substring $S[j..j+L[j]-1]$.

   > **Before we take any steps, our (empty) subsolution can be extended to an optimal solution. Assume that, after $i \geq 0$ steps, our subsolution can be extended to an optimal solution S. Then we show that after $i+1$ steps, our subsolution can be extended to an optimal solution S'. Therefore, by induction, we obtain an optimal solution.**

   Let $k = j + L[j]$. If S makes the $(i+1)$st substring $S[j..k-1]$, then $S' = S$. Because $S[j..k]$ doesn't occur in T, we know S can't make the $(i+1)$st substring longer than $L[j]$, so suppose S makes the $(i+1)$st substring $S[j..k'-1]$ with $k' < k$.
   Since $S[k..k+L[k]]$ doesn't occur in T, at some point S has to choose a substring ending at $S[\ell]$ in $S[k..k+L[k]-1]$. We make the next substring $S[k..\ell]$, and thereafter choose the rest of the substrings the same way S does. This way, we get a solution S' with the same number of substrings as S that extends our subsolution after $i+1$ steps.

4. According to `https://www150.statcan.gc.ca/t1/tbl1/en/tv.action?pid=1710000901`, the populations of Canada's provinces and territories in the first quarter of 2021 were as follows:

Newfoundland and Labrador	520,438
Prince Edward Island	159,819
Nova Scotia	979,449
New Brunswick	782,078
Quebec	8,575,944
Ontario	14,755,211
Manitoba	1,380,935
Saskatchewan	1,178,832
Alberta	4,436,258
British Columbia	5,153,039
Yukon	42,192
Northwest Territories	45,136
Nunavut	39,407

Suppose we choose a resident of Canada uniformly at random and let X be the province or territory where they live.

(a) Compute the entropy (in bits) of the random variable X.

(b) Compute $\sum_i p_i \lceil \lg(1/p_i) \rceil$, where p_i is the probability the resident of Canada lives in the ith province or territory listed above.

(c) Build a Huffman code for the probability distribution $p_1, \ldots, p_{13}$; what is its expected codeword length?

Solution: Since

$$\begin{aligned} & 520438 + 159819 + 979449 + 782078 + 8575944 + 14755211 + \\ & 1380935 + 1178832 + 4436258 + 5153039 + 42192 + 45136 + 39407 \quad = \quad 38048738 \end{aligned}$$

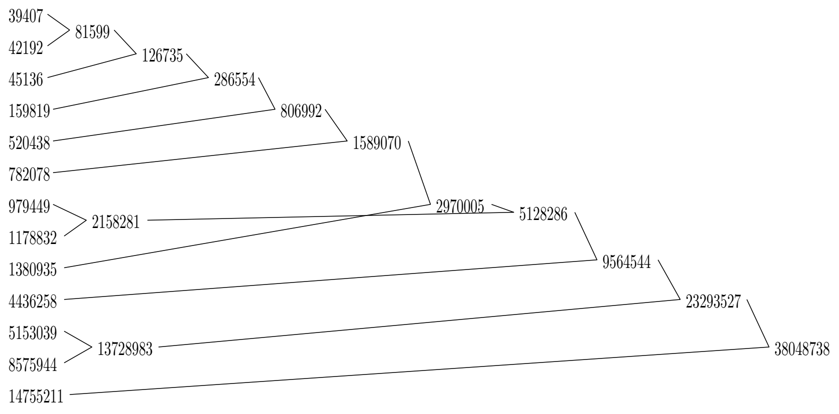
the entropy of X is

$$\begin{aligned} H(X) \quad = \quad & (520438/38048738) \lg(38048738/520438) + \\ & (159819/38048738) \lg(38048738/159819) + \\ & (979449/38048738) \lg(38048738/979449) + \\ & (782078/38048738) \lg(38048738/782078) + \\ & (8575944/38048738) \lg(38048738/8575944) + \\ & (14755211/38048738) \lg(38048738/14755211) + \\ & (1380935/38048738) \lg(38048738/1380935) + \\ & (1178832/38048738) \lg(38048738/1178832) + \\ & (4436258/38048738) \lg(38048738/4436258) + \\ & (5153039/38048738) \lg(38048738/5153039) + \\ & (42192/38048738) \lg(38048738/42192) + \\ & (45136/38048738) \lg(38048738/45136) + \\ & (39407/38048738) \lg(38048738/39407) \\ \approx \quad & 2.497\,. \end{aligned}$$

You can plug $(520438/38048738)\,\lg(38048738/520438)+\ldots$ straight into the Google search bar to evaluate it, but Google doesn't seem to recognize the ceiling function. For that, it's probably easiest to replace "lg" by "$*$ceiling((1/ log(2))$*$" and ")+" by "))+" and put an extra ")" at the end, and plug the resulting formula into a spreadsheet, to get

$$\begin{aligned}
& (520438/38048738) * \mathrm{ceiling}((1/\log(2)) * \log(38048738/520438)) + \\
& (159819/38048738) * \mathrm{ceiling}((1/\log(2)) * \log(38048738/159819)) + \\
& (979449/38048738) * \mathrm{ceiling}((1/\log(2)) * \log(38048738/979449)) + \\
& (782078/38048738) * \mathrm{ceiling}((1/\log(2)) * \log(38048738/782078)) + \\
& (8575944/38048738) * \mathrm{ceiling}((1/\log(2)) * \log(38048738/8575944)) + \\
& (14755211/38048738) * \mathrm{ceiling}((1/\log(2)) * \log(38048738/14755211)) + \\
& (1380935/38048738) * \mathrm{ceiling}((1/\log(2)) * \log(38048738/1380935)) + \\
& (1178832/38048738) * \mathrm{ceiling}((1/\log(2)) * \log(38048738/1178832)) + \\
& (4436258/38048738) * \mathrm{ceiling}((1/\log(2)) * \log(38048738/4436258)) + \\
& (5153039/38048738) * \mathrm{ceiling}((1/\log(2)) * \log(38048738/5153039)) + \\
& (42192/38048738) * \mathrm{ceiling}((1/\log(2)) * \log(38048738/42192)) + \\
& (45136/38048738) * \mathrm{ceiling}((1/\log(2)) * \log(38048738/45136)) + \\
& (39407/38048738) * \mathrm{ceiling}((1/\log(2)) * \log(38048738/39407)) \\
\approx\ & 3.132\,.
\end{aligned}$$

(Yes, you could also write a little program to compute it.)
A Huffman code with expected codeword length $97783314/38048738 \approx 2.570$ is shown below:



39407	$\times$ 10
42192	$\times$ 10
45136	$\times$ 9
159819	$\times$ 8
520438	$\times$ 7
782078	$\times$ 6
979449	$\times$ 5
1178832	$\times$ 5
1380935	$\times$ 5
4436258	$\times$ 3
5153039	$\times$ 3
8575944	$\times$ 3
14755211	$\times$ 1
97783314	

5. Suppose you have season passes for m train lines between n cities, with different expiry dates. A ticket lets you travel on the line between two cities as many times as you like, in either direction, from now until the ticket expires. How can you quickly determine the last date on which you will be able to reach any city from any other city using your passes?
   (a) Give a solution with the union-find data structure that takes $O(m\,\alpha(m,n))$ time after you've sorted the passes by expiry date.
   (b) Give a solution that colours and re-colours the cities, and takes $O(m)$ time after you've sorted the passes by expiry data.

You need not prove your solutions correct.

Solution: The key idea is to start with a graph whose n vertices are the cities and which contains no edges, and then insert the edges for the train lines in the non-increasing order of their tickets' expiry dates. The line whose insertion connects the graph is the one whose removal would disconnect the graph if we started with all the edges present and deleted them in non-decreasing order of their tickets' expiry dates.
Suppose we start with a singleton set for each city and, for each edge (u, v) we insert, use find queries on u and v to determine whether they're already in the same connected component and then a union operation on them if they are not. In $O(m\alpha(m, n))$ time we learn the last edge for which the find queries return distinct representatives, and thus whose insertion connects the graph.
Now suppose we start with all the cities coloured blue except for one, which is coloured red. Whenever we add an edge, if it is between two vertices with the same colour, we add it but do not re-colour anything; if it between a blue vertex and a red vertex, we start at the blue endpoint of that edge and re-colour red it and all the blue vertices it is currently connected to. The last edge for which we re-colour a blue vertex is the one whose insertion connects the graph. To see how this takes $O(m)$ time after the edges are sorted, consider that re-colouring a blue connected component takes time proportional to the number of edges already added to that component. Since we re-colour each vertex only one, in total we use $O(m)$ time.

Midterm 1 Solutions

1. For each cell (i, j) in the matrix below, write o, O, Θ, Ω or ω to indicate the relationship of the ith function on the left to the jth function along the top. If none of those relationships hold, leave the cell blank. Only the best answer possible will be considered correct (so writing O when the best answer is o doesn't count, for example). The cell $(1, 1)$ is filled in as an example: $n^{1/4} \in o(n^2)$, so that cell contains "o". **You need not explain your answers.**

	n^2	$((-1)^n + 1)n$	$3^{\lg n}$	$n^{\lg \lg n}$	$n \lg n$
$n^{1/4}$	o		o	o	o
2^n	ω	ω	ω	ω	ω
$\lceil \lg n \rceil !$	ω	ω	ω	o	ω
$n \sum_{j=1}^{n} (1/j)$	o	ω	o	o	Θ
$T(n) = 5T(n/4) + n^{3/2}$	o	ω	o	o	ω

2. Assuming you have an $O(n)$-time algorithm to find a separator of a planar graph on n vertices — that is, a subset of at most $2\sqrt{n}$ vertices after whose removal all remaining connected components each consist of at most $2n/3$ vertices — give a divide-and-conquer algorithm to find a minimum vertex cover of a planar graph on n vertices, similar to our algorithm for colouring a planar graph. Your mark will partly depend on how fast your algorithm is (although there is not believed to exist a polynomial-time algorithm). **You need not analyze your algorithm nor prove it correct.**

   Solution: We first find a separator S of the graph and then, for each subset T of S, we recursively find the minimum vertex cover of the graph that contains every vertex in T and none of the other vertices in S. To do this,
   (a) we ensure there are no edges between two vertices in $S \backslash T$ (if there are, then no vertex cover omits both);
   (b) for each vertex v in T, we add v to the vertex cover and delete all the edges incident to v;
   (c) for every remaining edge incident to a vertex u in $S \backslash T$, we add the other endpoint w of that edge to the vertex cover and delete all the edges incident to w;
   (d) we recurse on the remaining connected components.

   This takes time bounded by $T(n) = 2^{2\sqrt{n}}\, 2\, T(2n/3) \in O\left(2^{n^{1/2+\epsilon}}\right)$ for any positive constant ϵ — but you don't have to include an analysis.

3. Suppose we have a function that, given an unsorted sequence of n integers, in $O(n)$ time returns the (n/q)th, $(2n/q)$th, ..., $((q-1)n/q)$th smallest elements, called q-quantiles. Considering the time to compare elements to quantiles,
   (a) how quickly can we sort with this function when q is constant? **Solution:** $\Theta(n \log n)$ (in the comparison model, at least)
   (b) how quickly can we sort with this function when $q = \sqrt{n}$? **Solution:** the fastest I could figure out was still $\Theta(n \log n)$
   (c) if we can choose q freely, how should we choose it to sort as quickly as possible with this function? **Solution:** if we choose $q = n$ then we can sort in linear time

   You need not explain your answers.

(10% bonus question) How fast can we sort if the function takes $O(n)$ time and separates the integers in the array into $\sqrt{n}$ bins such that the ith bin contains the $((i-1)\sqrt{n}+1)$st through $(i\sqrt{n})$th smallest integers? (That is, the first bin contains the smallest $\sqrt{n}$ elements, the second bin contains the next $\sqrt{n}$ elements, etc.) **You need not explain your answer.**

Solution: The recurrence is

$$\begin{aligned}
T(n) &= n^{1/2}T(n^{1/2}) + O(n) \\
&= n^{1/2}(n^{1/4}T(n^{1/4}) + O(n^{1/2})) + O(n) \\
&= n^{\frac{1}{2}+\frac{1}{4}}T(n^{1/4}) + O(n) + O(n) \\
&= n^{\frac{1}{2}+\frac{1}{4}}(n^{1/8}T(n^{1/8}) + O(n^{1/4})) + O(n) + O(n) \\
&= n^{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}}T(n^{1/8}) + O(n) + O(n) + O(n) \\
&= \prod_{i=1}^{\lg\lg n}(n^{1/2^i} + O(n)) \\
&= O(n \lg\lg n)\,.
\end{aligned}$$

The trick here is to ask for what i we have $n^{1/2^i} = 2$ or, equivalently, $2^{2^i} = n$; taking lg twice on both sides, we get $i = \lg\lg n$. You don't need to write the recurrence or the explanation, though — writing $O(n \lg\lg n)$ is enough.

4. Imagine you're planning a post-lockdown canoe trip with friends, but
   - people want to bring different amounts of equipment,
   - everyone wants to be in the same canoe as their equipment,
   - you can have only so much equipment in each canoe (all the canoes are the same, and consider only the weight of the equipment),
   - any one person's equipment fits in one canoe,
   - everyone wants to row (so you can have at most two people in each canoe).

   You have a list of how much equipment each person wants to take (in kilos), and you know how much fits in a canoe. For example, if there are 3 people going and they want to take 37 kg, 52 kg and 19 kg of equipment and a canoe can hold up to 60 kg of equipment (plus up to 2 people), then you need at least 2 canoes: you can put the first and third people and their $37 + 19 = 56$ kg of equipment in one and the second person and their 52 kg in the other.
   Give a greedy algorithm to find the minimum number of canoes you need AND GIVE A PROOF OF CORRECTNESS!

   Solution: Let c be the capacity of a canoe in kilos. Sort the people into non-increasing order by the amount of equipment they want to take and consider them in that order. Suppose the ith person has w_i kilos of equipment. Pair them with the person who has the closest to $c - w_i$ kilos of equipment, without going over.

   > **Before we take any steps, our (empty) subsolution can be extended to an optimal solution. Assume that, after $i \geq 0$ steps, our subsolution can be extended to an optimal solution S. Then we show that after**

$i + 1$ steps, our subsolution can be extended to an optimal solution S'. Therefore, by induction, we obtain an optimal solution.[¶]

Suppose we pair the $(i+1)$st person with someone who has $w_j \leq c - w_{i+1}$ kilos of equipment. If S also has those people paired, then $S' = S$. If S doesn't have the $(i+1)$st person paired with anyone, then we can move the person with w_j kilos into the same boat as the $(i+1)$st person and obtain a solution S' that uses no more canoes than S and extends our subsolution after $i + 1$ steps. So, suppose S puts someone else in the canoe with the $(i+1)$st person, with $w_{j'}$ kilos of equipment. Since $w_{j'} \leq w_j$, we can swap the people with those amounts of equipment, to obtain a solution S' that uses no more canoes than S and extends our subsolution after $i + 1$ steps.

5. Give a greedy algorithm for Binary Knapsack that runs in $O(n \log n)$ time, where n is the number of items to consider, and achieves at least half the maximum profit when all the items have the same profit-to-weight ratio. Explain why your algorithm achieves this.

   Solution: We sort the items into non-decreasing order by profit (and, thus, also by weight). After we discard items that don't fit in the knapsack at all (even when it's empty), at least one of the follow statements is always true:
   - the knapsack is at least half full,
   - there is space for the next item,
   - there are no more items.

   If the knapsack is less than half full, then either it is completely empty (in which case any single remaining item fits), or the last item we put in took less than half the capacity, in which case the next item we consider takes less than half the capacity (because we're considering the items in non-decreasing order by weight). If we must stop because we've more than half-filled the knapsack, then we already have at least half the possible profit. If we must stop because we've run out of items, then we already have all the possible profit.

[¶]When I wrote before "Don't worry, this is the last time I'm going to write this", I meant it was the last time *that day*.

Assignment 5 Solutions

1. Write a program that takes two strings S and T and outputs an optimal alignment in $O(|S||T|)$ time, displayed as in the lecture notes. For example, if S = `AGATACATCA` and T = `GATTAGATACAT`, then an optimal alignment is

   ```
   AGAT-ACAT-CA-
   -GATTAGATACAT
   ```

   (I think). You need not analyze your algorithm nor prove it correct.

   Solution:

```
#include <stdio.h>
#include <stdlib.h>

#define MIN(a, b) (a < b ? a : b)
#define MIN3(a, b, c) (MIN(a, b) < c ? MIN(a, b) : c)

void align(char *S, char *T, int m, int n) {
  int A[m + 1][n + 1];
  char D[m + 1][n + 1];
  char *top, *bottom;

  A[0][0] = 0;
  D[0][0] = 's'; // we start at A[0][0]

  for (int i = 1; i <= m; i++) {
    A[i][0] = i;
    D[i][0] = 'v'; // we arrive at A[i][0] vertically
  }

  for (int j = 1; j <= n; j++) {
    A[0][j] = j;
    D[0][j] = 'h'; // we arrive at A[0][j] horizontally
  }

  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      A[i][j] = MIN3(A[i - 1][j] + 1, A[i][j - 1] + 1,
        A[i - 1][j - 1] + (S[i - 1] == T[j - 1] ? 0 : 1));

      if (A[i][j] == A[i - 1][j] + 1) {
        D[i][j] = 'v'; // we arrive at A[i][j] vertically
      } else if (A[i][j] == A[i][j - 1] + 1) {
        D[i][j] = 'h'; // we arrive at A[i][j] horizontally
      } else {
```

```
        D[i][j] = 'd'; // we arrive at A[i][j] diagonally
      }
    }
  }

  top = (char *) malloc(m + n + 1);
  bottom = (char *) malloc(m + n + 1);
  top[m + n] = '\0';
  bottom[m + n] = '\0';

  int i = m;
  int j = n;
  int p = m + n - 1;
  int q = m + n - 1;

  while (D[i][j] != 's') {
    if (D[i][j] == 'v') {
      top[p--] = S[i - 1];
      bottom[q--] = '-';
      i--;
    } else if (D[i][j] == 'h') {
      top[p--] = '-';
      bottom[q--] = T[j - 1];
      j--;
    } else {
      top[p--] = S[i - 1];
      bottom[q--] = T[j - 1];
      i--;
      j--;
    }
  }

  printf("%s\n%s\n", &top[p + 1], &bottom[q + 1]);

  return;
}

int main() {
  char *S = "AGATACATCA";
  char *T = "GATTAGATACAT";

  align(S, T, 10, 12);

  return(0);
}
```

2. Write a linear-time dynamic-programming algorithm for `https://leetcode.com/problems/maximum-subarray` and EXPLAIN IT.

   Solution:

```
int maxSubArray(int* nums, int numsSize){
    long int old, new, max;

    old = nums[0];
    max = nums[0];

    for (int i = 1; i < numsSize; i++) {
        if (old > 0) {
            new = old + nums[i];
        } else {
            new = nums[i];
        }

        if (new > max) {
            max = new;
        }

        old = new;
    }

    return(max);
}
```

   The sum of the max-sum non-empty subarray ending at `nums[0]` is just `nums[0]`. If the sum of the max-sum non-empty subarray ending at `nums[i - 1]` is positive, then the sum of the max-sum non-empty subarray ending at `nums[i]` is the sum of the max-sum non-empty subarray ending at `nums[i - 1]` plus `nums[i]`; otherwise, it's just `nums[i]`.

   (I realized after posting this assignment that the terrible explanations I was remembering were for the linear-time solutions to Question 4, which don't use a range-max data structure. Sorry for the confusion!)

3. Suppose your professor has assigned a "profit" to each of several indivisible food items, expressing how much he likes each item. He's now filling his knapsack and trying to select items to maximize the total profit. The food items are light but bulky, so the key constraint now is the *volume* the knapsack can hold, rather than the *weight*. Even though the food items cannot be cut, they can be *squashed*, which reduces their volume by a factor of 2 — but also reduces their profit by a factor of 2 (since squashed food is not as appetizing).

   Write a dynamic-programming algorithm that runs in time polynomial in the number of items and the capacity of the knapsack (in litres) and tells your professor which food items to select and, of the selected ones, which ones to squash. You can assume the knapsack's capacity and the original volume in litres of each item are integers. Explain why your algorithm is correct.

Solution: Suppose there are n items and let v_i and p_i be the volume and profit of the unsquashed ith item, for $1 \leq i \leq n$. Let c capacity of the knapsack. For $0 \leq i \leq n$ and $0 \leq j \leq 2c$, we want to compute the maximum profit $A[i, j]$ we can get from the first i items with volume exactly $j/2$.
When computing $A[i, j]$, we can either take the ith item unsquashed, in which case $A[i, j] = A[i-1, j-2v_i]+p_i$ (considering $A[i, j] = -\infty$ for $j < 0$); or we can take it squashed, in which case $A[i, j] = A[i-1, j-v_i] + p_i/2$; or we can leave it, in which case $A[i, j] = A[i-1, j]$.
(Again, I should have just asked for the optimal profit, not the actual list of items to take and squash.)

```
A [i, 0] = 0 for i from 0 to n
A [0, j] = -infty for j from 1 to 2 c
B [i, 0] = "leave" for i from 0 to n
B [0, j] = "?!" for j from 1 to 2 c

for i from 1 to n
  for j from 1 to 2 c
    A[i, j] = max(A [i - 1, j - 2 v_i] + p_i,
                  A [i - 1, j - v_i] + p_i / 2,
                  A [i, j] = A[i - 1, j])
    if (A[i, j] == A[i - 1, j - 2 v_i] + p_i)
      B [i, j] = "take"
    else if (A [i, j] == A [i - 1, j - v_i] + p_i / 2)
      B [i, j] = "squash"
    else
      B [i, j] = "leave"
    end if
  end for
end for

let j be the column with the maximum value in A[n, 0..2 c]

for i from n to 1
  print B [i, j] " item " i "\n"
  if B [i, j] == "take"
    j = j - 2 v_i
  else if B [i, j] == "squash"
    j = j - v_i
  end if
end for
```

4. Write pseudo-code — you don't have to code this — for an $O(n \log n)$-time algorithm that takes a sequence of n integers and finds the longest *slowly* increasing subsequence (LSIS), where an LSIS is a sequence in which each number after the first is larger than it's predecessor but not by more than 10. Explain why your algorithm is correct.

   Solution: Let $x_1, \ldots, x_n$ be the sequence of integers.
   To find the length of the LSIS of $x_1, \ldots, x_n$, we maintain the invariant that, when processing x_i, we have inserted into a dynamic range-max data structure each point (x_h, y_h) for $h < i$, where y_h is the length of the LSIS of $x_1, \ldots, x_h$ that includes x_h.
   The length of the LSIS of $x_1, \ldots, x_i$ that includes x_i, is the length of the LSIS of $x_1, \ldots, x_{i-1}$ that ends at a values x_h with $x_i - 10 \leq x_h < x_i$. Therefore, we query the range-max data structure with $[x_i - 10, x_i)$. If it returns (x_h, y_h), then we insert $(x_i, y_h + 1)$ into the data structure. If it doesn't return a point, then there are no entries in $x_1, \ldots, x_{i-1}$ in the desired range, and we insert $(x_i, 1)$.
   After processing x_n, we query the data structure with $[-\infty, +\infty]$; the y-component of the returned point is the length of the LSIS of $x_1, \ldots, x_n$.
   To find the LSIS itself, and not just its length, we can either
   - when inserting the point $(x_i, y_h + 1)$, store with it as satellite data a pointer to the point (x_h, y_h) from which we calculated $y_h + 1$ (and store a null pointer with $(x_i, 1)$);
   - when inserting the point $(x_i, y_h + 1)$, insert the key $(x_i, y_h + 1)$ into an $O(\log n)$-time dynamic dictionary data structure (such as an AVL tree) with (x_h, y_h) as satellite data (and store (NULL, NULL) with $(x_i, 1)$).

   (I should have asked only for the length; finding the actual LSIS is optional.)

```
RMQ_structure Q
dictionary D

for i from 1 to n
  (x_h, y_h) = Q.query [x_i - 10, x_i)
  if (y_h != NULL)
    Q.insert (x_i, y_h + 1)
    D.insert (x_i, y_h + 1) with (x_h, y_h) as satellite data
  else
    Q.insert (x_i, 1)
    D.insert (x_i, 1) with NULL as satellite data
  end if
end for

(x_h, y_h) = Q.query [-infty, +infty]
print x_h
while x_h != NULL
  (x_h, y_h) = satellite data of D.query (x_h, y_h)
  print x_h
end while
```

5. Modify the code in the lecture notes for building an optimal binary search tree such that it runs in $O(n^2)$ time instead of $O(n^3)$ time. You need not analyze your algorithm nor prove it correct.

Solution:

```
#include <stdio.h>

#define MIN(a, b) (a < b ? a : b)
#define MAX(a, b) (a > b ? a : b)

int optBST(int *F, int n) {
  int A[n][n];
  int R[n][n];
  int S[n + 1];

  S[0] = 0;
  for (int i = 0; i < n; i++) {
    A[i][i] = F[i];
    R[i][i] = i;
    S[i + 1] = S[i] + F[i];
  }

  for (int size = 2; size <= n; size++) {
    for (int i = 0; i < n - size + 1; i++) {
      int j = i + size - 1;

      A[i][j] = A[i + 1][j];
      R[i][j] = i;

      for (int r = MAX(R[i][j - 1], i + 1); r <= MIN(R[i + 1][j], j - 1); r++) {
        if (A[i][r - 1] + A[r + 1][j] < A[i][j]) {
          A[i][j] = A[i][r - 1] + A[r + 1][j];
          R[i][j] = r;
        }
      }

      if (A[i][j - 1] < A[i][j]) {
        A[i][j] = A[i][j - 1];
        R[i][j] = j;
      }

      A[i][j] += S[j + 1] - S[i];
    }
  }

  // printing the matrix is optional
  fprintf(stderr, "\t");
  for (int j = 0; j < n; j++) {
    fprintf(stderr, "\t%i", j);
  }
  fprintf(stderr, "\n");

  for (int i = 0; i < n; i++) {
    fprintf(stderr, "\t%i", i);
    for (int j = 0; j < n; j++) {
      fprintf(stderr, "\t");
      if (j >= i) {
        fprintf(stderr, "%i", A[i][j]);
      }
    }
    fprintf(stderr, "\n");
  }

  return(A[0][n - 1]);
}

int main() {
  int F[] = {5, 3, 4, 1, 1, 2, 1};

  printf("%i\n", optBST(F, 7));
  return(0);
}
```

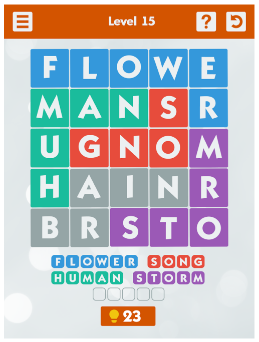
Assignment 6 Solutions

1. Show that Subset Sum is *self-reducible*: that is, if you have an algorithm that, given a set S of integers and a target t, in polynomial time (in the cardinality of S) determines whether a subset of S sums to t, then you can use it to design an algorithm that, given S and t, in polynomial time (in the cardinality of S) returns such a subset.

   Solution: Choose a number $x \in S$ and check if $S - \{x\}$ has a subset that sums to $t - x$; if so, add x to the solution and recursively find a subset of $S - \{x\}$ that sums to $t - x$; otherwise, discard x and recursively find a subset of $S - \{x\}$ that sums to t.

2. A *grid graph* is a graph whose vertices are labelled with distinct pairs of integers such that a vertex u labelled (x, y) is adjacent to a vertex v if and only if v is labelled $(x-1, y)$, $(x+1, y)$, $(x, y-1)$ or $(x, y+1)$. HamPath is known to be NP-complete even when restricted to grid graphs.
   For the problem Wordz 2, we are given an $n \times n$ grid of characters, a dictionary of strings and an integer k and asked if the grid contains at least k sequences of characters such that
   - if a character in a sequence has coordinates (x, y), then the next character must have coordinates $(x-1, y)$, $(x+1, y)$, $(x, y-1)$ or $(x, y+1)$,
   - each character in the grid appears at most once across all of the sequences,
   - each sequence is a distinct string in the dictionary.

   The image below shows an instance of Wordz 2 with a 5×5 grid and the dictionary of English words, with 4 sequences indicated with blue, red, green and purple. (The grey characters are not a sequence, although they could be.)
   Give a polynomial-time reduction from HamPath on Grid Graphs to Wordz 2. (You can assume all the coordinates in the vertices' labels in the graph are polynomial in the number of vertices.)



   Solution: Suppose we are given an instance G of HamPath on Grid Graphs on n vertices. Notice that if the difference between the maximum and minimum x-coordinates is more than $n - 1$, or the difference between the maximum and minimum y-coordinates is, then there can be no Hamiltonian path (so, trivially, there is one if and only if an instance of Wordz 2 consisting of a grid with a single letter N, a dictionary containing a single string Y and $k = 1$).

Otherwise, we shift all the x-coordinates by the same amount so the smallest is 1, and all the y-coordinates by the same amount so the smallest is 1, and then create an $n \times n$ grid in which the cell (i, j) contains a Y if there is a vertex now labelled (i, j), and an N otherwise. We create a dictionary with one string, consisting of n copies of Y. That string corresponds to a non-self-crossing path in the grid if and only if G has a Hamiltonian path.

3. For the problem Integer Linear Programming (ILP) we are given a set of linear constraints such as

$$\begin{aligned} 3x_1 + 5x_2 &\leq 5 \\ 4x_2 - 2x_3 &= 10 \\ 6x_1 - x_2 + 3x_3 &\geq 6 \end{aligned}$$

   and asked if there is a solution with all of the variables' values integers. Give a polynomial-time reduction from 3-Sat to ILP.

   Solution: Suppose we are given a 3-Sat formula F on n variables with m clauses. For each variable x in F we create variables x_+ and x_- in the ILP, with the constraints
   - $-1 < x_+ < 2$
   - $-1 < x_- < 2$
   - $x_+ + x_- < 2$

   so that any solution to the IPL must set exactly one of x_+ and x_- to 1 and the other to 0. For each clause in F we add a constraint saying the sum of the variables in the ILP that correspond to the literals in that clause sum more than 0, so that one of them must be 1. For example, for the clause $(x \vee \neg y \vee z)$, we add the constraint $x_+ + y_- + z_+ > 0$. Building the IPL takes polynomial time and there is a solution to it if and only if F is satisfiable.

4. We can reduce 3-Col to Planar 3-Col in polynomial time. Why doesn't our $3^{O(n^{1/2} \log n)}$-time divide-and-conquer algorithm for Planar 3-Col give us a $3^{O(n^{1/2} \log n)}$-time algorithm for 3-Col?

   Solution: The reduction from 3-Col to Planar 3-Col increases the number of vertices (see Figure 13.5, and the size n' of the resulting instance of Planar 3-Col can be sufficiently larger than the size n of the original instance of 3-Col that $3^{O(n'^{1/2} \log n)} \subset \omega(3^n n)$. (It's enough to answer that n' can be bigger than n.)

5. We saw a 2-approximation algorithm for the search version of Vertex Cover, and the complement of a vertex cover is an independent set; does that mean we have a 2-approximation algorithm for the search version of Independent Set? Why or why not?

   Solution: Suppose a graph G on n vertices has a minimum vertex cover of size $3n/8$, so its maximum independent set has size $5n/8$. A 2-approximation for vertex cover can return a subset of size $3n/4$, the complement of which is an independent set of size $n/4$ — which is less than half as big as the maximum independent set.

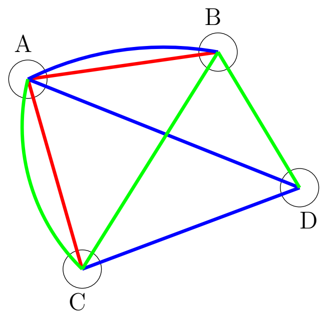
Midterm 2 Solutions

1. Suppose your class is on a field trip to an island with n towns on it, connected by m town-to-town buses (which run in both directions), run by c companies. Each company's buses are a different colour, and there can be buses from two or more companies running between two towns. You have a map showing which companies run buses between which towns. The drivers have a relaxed attitude to schedules and the buses run often, so there's no telling which buses will be arriving and leaving next.

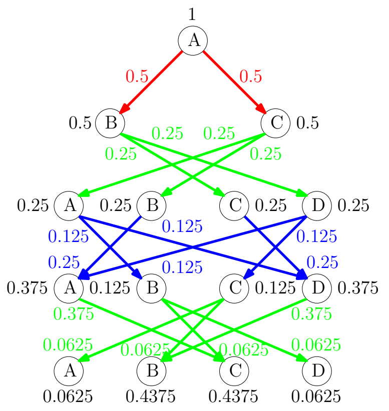
   Your classmate Ryan has wandered off and got lost and you're (somewhat reluctantly) trying to find him. You'd told him which buses the class was supposed to take during the day, and given him tickets from the appropriate companies, the same colours as the buses and stapled together in the right order. Ryan didn't remember which towns the class was going to visit, however, so he always took the first bus he saw of the colour of the next ticket, tearing off that ticket and giving it to the driver.

   Design a polynomial-time dynamic-programming algorithm that, given the map of the bus routes, Ryan's starting point and the colours of the buses he took, calculates the probability Ryan is in each of the n towns.

   For example, if the map is as shown below on the left and Ryan started in town A and took a red bus, a green bus, a blue bus and another green bus, then his itinerary could be any of those shown below in the center, and the probability of him being in a certain town after a certain number of steps and of taking trips between cities is as shown in the DAG below on the right.



A-B-C-D-B,
A-B-D-A-C,
A-B-D-C-A,
A-B-D-C-B,
A-C-A-B-C,
A-C-A-B-D,
A-C-A-D-B,
A-C-B-A-C .



   The probability Ryan went first from A to B is 0.5, and the probability he went first from A to C is 0.5. Therefore, after one trip, the probability he was in B is 0.5 and the probability he was in C is 0.5.

   The probability Ryan's second trip took him from B to C is 0.5 times the probability he was in B, or 0.25. The probability it took him from B to D is also 0.5 times the probability he was in B, or 0.25. The probability it took him from C to A is 0.5 times the probability he was in C, or 0.25. The probability it took him from C to B is 0.5 times the probability he was in C, or 0.25. Therefore, after two trips, the probability is 0.25 he was in any particular town.

The probability Ryan's third trip took him from A to B is 0.5 times the probability he was in A, or 0.125. The probability it took him from A to D is 0.5 times the probability he was in A, or 0.125. The probability it took him from B to A is the probability he was in A, or 0.25. The probability it took him from C to D is the probability he was in C, or 0.25. The probability it took him from D to A is 0.5 times the probability he was in D, or 0.125. The probability it took him from D to C is 0.5 times the probability he was in D, or 0.125.

Therefore, after three trips, the probabilities Ryan was in A, B, C, D are, respectively, $0.25 + 0.125 = 0.375$ (the probability his third trip took him from B to A plus the probability it took him from D to A), 0.125, 0.125 and $0.125 + 0.25 = 0.375$ (the probability his third trip took him from A to D plus the probability it took him from C to D).

You can compute the probability of Ryan being in a particular town after four trips similarly. Isn't it lucky you're on an island, so you can't accidentally lose Ryan forever?

(Hint: first design an algorithm that computes the number of ways Ryan could have ended up in a town, and then modify it to compute the probability.)

Solution: The example didn't show how to deal with situations such as Ryan starting in town A of the map shown, and then taking a blue bus and a red bus. In that case, he can only take both trips by travelling to B and then back to A; if he takes a blue bus from A to D, then he's stuck there.

We can still use the dynamic program suggested by the example, but we should normalize the distribution we get at the end (because the question says "given ... the colours of the buses he took", so we can assume he didn't get stuck partway through the sequence). Not normalizing costs 20% of the mark.

Suppose there are b tickets. We fill an array $M[0..b, 1..n]$, starting by setting the entry of $M[0, 1..n]$ corresponding to Ryan's starting point to 1 and the other entries in M to 0. For i from 1 to b and j from 1 to n, we take the value in $M[i-1, j]$, divide it by the number of ways to leave the jth town on a bus with the ith colour and, for each of those ways, add the quotient to $M[i, j']$ where j' is the number of the town reached by that bus. When we've propagated the values to $M[b, 1..n]$, we normalize the distribution in that row.

To see why this is the correct distribution, suppose we let millions of Ryans wander. Some may get stranded along the way and lost, alas, but of those who manage to make all b trips, the fraction arriving at each town after those b trips gives us an idea of the probability a single Ryan will end up there. Normalizing gives us that fraction.

2. Your professor Travis told your TA Sarah that he was going to ask you to modify the solution to the alignment question on Assignment 5, to compute an optimal alignment using only one pass through the matrix.[‖] In contrast, that assignment question allows filling in the matrix and then walking back from the bottom right corner to the top left corner to compute the alignment.

   Travis claimed it was possible by keeping two more arrays, top$[0..m, 0..n]$ and bottom$[0..m, 0..n]$, where top$[i, j]$ is a pointer to a string of length at most $m + n + 1$ (including the end-of-string delimiter) containing the top line in an optimal alignment of $S[1..i]$ to $T[1..j]$, and bottom$[i, j]$ is a pointer to a string of length at most $m+n+1$ containing the bottom line in that alignment.

[‖] Yes, this question is based on a true story.

To compute top$[i, j]$, we `sprintf` into an empty string either top$[i-1, j]$ or top$[i-1, j-1]$ or top$[i, j-1]$, followed by either $S[i-1]$ or '`-`'. To compute bottom$[i, j]$, we `sprintf` into an empty string either bottom$[i-1, j]$ or bottom$[i-1, j-1]$ or bottom$[i, j-1]$, followed by either $T[j-1]$ or `-`.

Sarah correctly pointed out that `malloc`ing and `sprintf`ing a string of length $\Omega(m+n+1)$ takes $\Omega(m+n+1)$ time, so Travis's solution takes cubic time. Help Travis by figuring out how to modify his solution (shown below) to use quadratic time again.

(Hint: pointers are your friends!)

```
#include <stdio.h>
#include <stdlib.h>

#define MIN(a, b) (a < b ? a : b)
#define MIN3(a, b, c) (MIN(a, b) < c ? MIN(a, b) : c)

void align(char *S, char *T, int m, int n) {
  int A[m + 1][n + 1];
  char *top[m + 1][n + 1];
  char *bottom[m + 1][n + 1];

  A[0][0] = 0;
  top[0][0] = (char *) malloc(m + n + 1);
  bottom[0][0] = (char *) malloc(m + n + 1);
  sprintf(top[0][0], "");
  sprintf(bottom[0][0], "");

  for (int i = 1; i <= m; i++) {
    A[i][0] = i;
    top[i][0] = (char *) malloc(m + n + 1);
    bottom[i][0] = (char *) malloc(m + n + 1);
    sprintf(top[i][0], "%s%c", top[i - 1][0], S[i - 1]);
    sprintf(bottom[i][0], "%s-", bottom[i - 1][0]);
  }

  for (int j = 1; j <= n; j++) {
    A[0][j] = j;
    top[0][j] = (char *) malloc(m + n + 1);
    bottom[0][j] = (char *) malloc(m + n + 1);
    sprintf(top[0][j], "%s-", top[0][j - 1]);
    sprintf(bottom[0][j], "%s%c", bottom[0][j - 1], T[j - 1]);
  }

  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      A[i][j] = MIN3(A[i - 1][j] + 1, A[i][j - 1] + 1,
        A[i - 1][j - 1] + (S[i - 1] == T[j - 1] ? 0 : 1));
      top[i][j] = (char *) malloc(m + n + 1);
      bottom[i][j] = (char *) malloc(m + n + 1);

      if (A[i][j] == A[i - 1][j] + 1) {
        sprintf(top[i][j], "%s%c", top[i - 1][j], S[i - 1]);
        sprintf(bottom[i][j], "%s-", bottom[i - 1][j]);
      } else if (A[i][j] == A[i][j - 1] + 1) {
        sprintf(top[i][j], "%s-", top[i][j - 1]);
        sprintf(bottom[i][j], "%s%c", bottom[i][j - 1], T[j - 1]);
      } else {
        sprintf(top[i][j], "%s%c", top[i - 1][j - 1], S[i - 1]);
        sprintf(bottom[i][j], "%s%c", bottom[i - 1][j - 1], T[j - 1]);
      }
    }
  }

  printf("%s\n%s\n", top[m][n], bottom[m][n]);

  return;
}

int main() {
  char *S = "AGATACATCA";
  char *T = "GATTAGATACAT";

  align(S, T, 10, 12);

  return(0);
}
```

Solution: The key idea is to use linked lists instead of strings, as in the code below. You can describe your solution precisely instead of giving code!

```
#include <stdio.h>
#include <stdlib.h>

#define MIN(a, b) (a < b ? a : b)
#define MIN3(a, b, c) (MIN(a, b) < c ? MIN(a, b) : c)

typedef struct node {
  char letter;
  struct node *pointer;
} node;
```

```
void printList(node *L) {
  if (L != NULL) {
    printList(L -> pointer);
    printf("%c", L -> letter);
  }
  return;
}

void align(char *S, char *T, int m, int n) {
  int A[m + 1][n + 1];
  node *top[m + 1][n + 1];
  node *bottom[m + 1][n + 1];

  A[0][0] = 0;
  top[0][0] = NULL;
  bottom[0][0] = NULL;

  for (int i = 1; i <= m; i++) {
    A[i][0] = i;
    top[i][0] = (node *) malloc(sizeof(node));
    top[i][0] -> letter = S[i - 1];
    top[i][0] -> pointer = top[i - 1][0];
    bottom[i][0] = (node *) malloc(sizeof(node));
    bottom[i][0] -> letter = '-';
    bottom[i][0] -> pointer = bottom[i - 1][0];
  }

  for (int j = 1; j <= n; j++) {
    A[0][j] = j;
    top[0][j] = (node *) malloc(sizeof(node));
    top[0][j] -> letter = '-';
    top[0][j] -> pointer = top[0][j - 1];
    bottom[0][j] = (node *) malloc(sizeof(node));
    bottom[0][j] -> letter = T[j - 1];
    bottom[0][j] -> pointer = bottom[0][j - 1];
  }

  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      A[i][j] = MIN3(A[i - 1][j] + 1, A[i][j - 1] + 1,
        A[i - 1][j - 1] + (S[i - 1] == T[j - 1] ? 0 : 1));

      top[i][j] = (node *) malloc(sizeof(node));
      bottom[i][j] = (node *) malloc(sizeof(node));
      if (A[i][j] == A[i - 1][j] + 1) {
        top[i][j] = (node *) malloc(sizeof(node));
        top[i][j] -> letter = S[i - 1];
        top[i][j] -> pointer = top[i - 1][j];
        bottom[i][j] = (node *) malloc(sizeof(node));
        bottom[i][j] -> letter = '-';
        bottom[i][j] -> pointer = bottom[i - 1][j];
      } else if (A[i][j] == A[i][j - 1] + 1) {
        top[i][j] = (node *) malloc(sizeof(node));
        top[i][j] -> letter = '-';
        top[i][j] -> pointer = top[i][j - 1];
        bottom[i][j] = (node *) malloc(sizeof(node));
        bottom[i][j] -> letter = T[j - 1];
        bottom[i][j] -> pointer = bottom[i][j - 1];
      } else {
        top[i][j] = (node *) malloc(sizeof(node));
        top[i][j] -> letter = S[i - 1];
        top[i][j] -> pointer = top[i - 1][j - 1];
        bottom[i][j] = (node *) malloc(sizeof(node));
        bottom[i][j] -> letter = T[j - 1];
        bottom[i][j] -> pointer = bottom[i - 1][j - 1];
      }
    }
  }

  printList(top[m][n]);
  printf("\n");
  printList(bottom[m][n]);
  printf("\n");
  return;
}

int main() {
  char *S = "AGATACATCA";
  char *T = "GATTAGATACAT";

  align(S, T, 10, 12);

  return(0);
}
```

Bonus (worth 10% of the midterm): Can you reduce the space usage to $O((m+n)k)$, assuming you're given the edit distance k between S and T?

(Hint: banded dynamic programming and garbage collections are your friends!)

Solution: The first part of the solution is not to keep the whole two-dimensional arrays A, `top` and `bottom` at once since, of each one, we need only the row we are currently computing and the preceding row. The linked lists we're still using will not be lost, because we still have pointers to them; the others will be handled by garbage collection. The second part of the solution is not to compute alignment costs or lists for pairs of strings whose lengths differ by more than k. This way, at any point we have pointers to $O(k)$ `top` lists and $O(k)$ `bottom` lists, each of length at most $m + n$, so they take $O((m + n)k)$ space.

```
#include <stdio.h>
#include <stdlib.h>

#define MIN(a, b) (a < b ? a : b)
#define MIN3(a, b, c) (MIN(a, b) < c ? MIN(a, b) : c)
#define MAX(a, b) (a > b ? a : b)

typedef struct node {
  char letter;
  struct node *pointer;
} node;

void printList(node *L) {
  if (L != NULL) {
    printList(L -> pointer);
    printf("%c", L -> letter);
  }
  return;
}

void align(char *S, char *T, int m, int n, int k) {
  int *A[m + 1];
  node **top[m + 1];
  node **bottom[m + 1];

  A[0] = (int *) malloc((n + 1) * sizeof(int));
  top[0] = (node **) malloc((n + 1) * sizeof(node *));
  bottom[0] = (node **) malloc((n + 1) * sizeof(node *));

  A[0][0] = 0;
  top[0][0] = NULL;
  bottom[0][0] = NULL;

  for (int j = 1; j <= n; j++) {
    A[0][j] = j;
    top[0][j] = (node *) malloc(sizeof(node));
    top[0][j] -> letter = '-';
    top[0][j] -> pointer = top[0][j - 1];
    bottom[0][j] = (node *) malloc(sizeof(node));
    bottom[0][j] -> letter = T[j - 1];
    bottom[0][j] -> pointer = bottom[0][j - 1];
  }

  for (int i = 1; i <= m; i++) {
    A[i] = (int *) malloc((n + 1) * sizeof(int));
    top[i] = (node **) malloc((n + 1) * sizeof(node *));
    bottom[i] = (node **) malloc((n + 1) * sizeof(node *));

    A[i][0] = i;
    top[i][0] = (node *) malloc(sizeof(node));
    top[i][0] -> letter = S[i - 1];
    top[i][0] -> pointer = top[i - 1][0];
    bottom[i][0] = (node *) malloc(sizeof(node));
    bottom[i][0] -> letter = '-';
    bottom[i][0] -> pointer = bottom[i - 1][0];

    if (i - k >= 0) {
      A[i][i - k - 1] = m + n + 1;
    }

    if (i + k <= n) {
      A[i - 1][i + k] = m + n + 1;
    }

    for (int j = MAX(i - k, 1); j <= MIN(i + k, n); j++) {
      A[i][j] = MIN3(A[i - 1][j] + 1, A[i][j - 1] + 1,
        A[i - 1][j - 1] + (S[i - 1] == T[j - 1] ? 0 : 1));

      top[i][j] = (node *) malloc(sizeof(node));
      bottom[i][j] = (node *) malloc(sizeof(node));

      if (A[i][j] == A[i - 1][j] + 1) {
        top[i][j] = (node *) malloc(sizeof(node));
        top[i][j] -> letter = S[i - 1];
        top[i][j] -> pointer = top[i - 1][j];
        bottom[i][j] = (node *) malloc(sizeof(node));
        bottom[i][j] -> letter = '-';
        bottom[i][j] -> pointer = bottom[i - 1][j];
      } else if (A[i][j] == A[i][j - 1] + 1) {
        top[i][j] = (node *) malloc(sizeof(node));
        top[i][j] -> letter = '-';
        top[i][j] -> pointer = top[i][j - 1];
        bottom[i][j] = (node *) malloc(sizeof(node));
        bottom[i][j] -> letter = T[j - 1];
        bottom[i][j] -> pointer = bottom[i][j - 1];
      } else {
        top[i][j] = (node *) malloc(sizeof(node));
        top[i][j] -> letter = S[i - 1];
        top[i][j] -> pointer = top[i - 1][j - 1];
        bottom[i][j] = (node *) malloc(sizeof(node));
        bottom[i][j] -> letter = T[j - 1];
        bottom[i][j] -> pointer = bottom[i - 1][j - 1];
      }
    }

    free(A[i - 1]);
    free(top[i - 1]);
    free(bottom[i - 1]);
  }

  printList(top[m][n]);
  printf("\n");
  printList(bottom[m][n]);
  printf("\n");
  return;
}

int main() {
  char *S = "AGATACATCA";
  char *T = "GATTAGATACAT";

  align(S, T, 10, 12, 6);

  return(0);
}
```

3. For the problem Longest Kind-Of Increasing Subsequence (LKOIS), we're given a sequence $S[1..n]$ of integers and asked to find the longest subsequence S' of S such that $S'[i-1] - 3 \leq S'[i]$ for $1 < i \leq |S'|$. Give an $O(n \log n)$ algorithm for LKOIS.

   Solution: To find the length of the LKOIS of S, we maintain the invariant that, when processing $S[i]$, we have inserted into a dynamic range-max data structure each point $(S[h], y_h)$ for $h < i$, where y_h is the length of the LKOIS of $S[1..h]$ that includes $S[h]$.

   The length of the LKOIS of $S[1..i]$ that includes $S[i]$, is the length of the LKOIS of $S[1..i-1]$ that ends at a value $S[h]$ with $S[h] - 3 \leq S[i]$. Therefore, we query the range-max data structure with $[-\infty, S[i] + 3)$. If it returns $(S[h], y_h)$, then we insert $(S[i], y_h + 1)$ into the data structure. If it doesn't return a point, then there are no entries in $S[1..i-1]$ in the desired range, and we insert $(S[i], 1)$.

After processing $S[n]$, we query the data structure with $[-\infty, +\infty]$; the y-component of the returned point is the length of the LKOIS of S.

To find the LSIS itself, and not just its length, we can either

- when inserting the point $(S[i], y_h + 1)$, store with it as satellite data a pointer to the point $(S[h], y_h)$ from which we calculated $y_h + 1$ (and store a null pointer with $(S[i], 1)$);
- when inserting the point $(S[i], y_h + 1)$, insert the key $(S[i], y_h + 1)$ into an $O(\log n)$-time dynamic dictionary data structure (such as an AVL tree) with $(S[h], y_h)$ as satellite data (and store (NULL, NULL) with $(S[i], 1)$).

4. For the problem Partition, we're given a set S of positive integers that sums to $2t$ and asked if there is a subset of S that sums to exactly t. Prove Partition is NP-complete by
   - showing Partition is in NP,
   - reducing one of the NP-complete problems we've seen in class to Partition.

   Solution: Partition is in NP because, given a set S' of integers, in polynomial time we can check that $S' \subset S$ and the sum of S' is T.

   We reduce Subset Sum with only positive integers to Partition. We know Subset Sum with only positive integers is NP-complete because our reduction from 3-Sat to Subset Sum maps formulas to instances of Subset Sum with only positive integers.

   (There is no penalty for considering Subset Sum with only positive integers with no explanation of why it is NP-complete, or even with no explicit mention that all the integers are assumed to be positive.)

   Assume we are given a set X of positive integers $x_1, \ldots, x_m$ and a target n, and asked if a subset of X sums to n. Let h be half the sum of X. We add two numbers x_{m+1} and x_{m+2} to X, where $x_{m+1} = 5h - n$ and $x_{m+2} = 3h + n$.

   Suppose there is a subset of X that sums to n. Then that subset plus x_{m+1} sums to $5h$. The rest of X sums to $2h - n$, so the rest of X plus x_{m+2} also sums to $5h$.

   Now suppose there is a subset of $X \cup \{x_{m+1}, x_{m+2}\}$ that sums to half the sum of $X \cup \{x_{m+1}, x_{m+2}\}$, that is, $(2h + (5h - n) + (3h + n))/2 = 5h$. If that subset includes $x_{m+1} = 5h - n$, then the rest of that subset sums to n. If that subset includes $x_{m+2} = 3h + n$, then the rest of that subset sums to $2h - n$. Since $2h - n < 5h - n$, the rest of that subset does not include x_{m+1}; therefore, it is a subset of X which, when removed, leaves a subset that sums to $2h - (2h - n) = n$.

5. Write a program that, given a list of the edges in a connected graph G on the vertices $1, \ldots, n$, in polynomial time outputs a Boolean formula F that is satisfiable if and only if G has a Hamiltonian path. You can assume the list of edges looks something like

   ```
   (1, 2)
   (1, 3)
   (4, 2)
   (6, 5)
   (5, 3)
   ```

   with one pair per line, and your output should consist of a single line containing copies of space, `(`, `)`, `AND`, `OR`, `NOT` and variables that look something like `x1`, `x2`, etc.

   Solution: This is actually discussed in one of the lectures. I should have made life easier for everyone by saying n is given at the beginning of the input, and that you can use variables of the form `x_{i, j}` . You can convert a variable in that form into a variable with a single number by writing `xi` where `i` is `i * (n + 1) + j`, but it's annoying. I told someone who asked that you can use two numbers, and we won't look at it while marking. Actually, I've used two numbers in the code below, because it makes the output more intelligible.

The idea is to write subformulas saying that every vertex appears in the path somewhere, every position in the path has a vertex associated with it, the same vertex can appear in two positions, two vertices can't appear in the same position (I think this is actually redundant — if follows from the previous constraints?), and if there isn't an edge between u and v then if u appears jth in the path then v can't appear $(j+1)$st. (The `true` is there because add that is simpler than getting rid of the last AND.)

```
#include <stdio.h>
#include <stdlib.h>

#define MAX(a, b) (a > b ? a : b)

typedef struct node {
  int u;
  int v;
  struct node *pointer;
} node;

int main() {
  node *L = NULL;
  int u, v, n = 0;

  while (scanf("(%i, %i)\n", &u, &v) != 0) {
    node *temp = (node *) malloc(sizeof(node));
    temp -> u = u;
    temp -> v = v;
    temp -> pointer = L;
    L = temp;

    n = MAX(n, u);
    n = MAX(n, v);
  }

  int M[n + 1][n + 1];

  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      M[i][j] = 0;
    }
  }

  while (L != NULL) {
    u = L -> u;
    v = L -> v;
    M[u][v] = 1;
    M[v][u] = 1;
    L = L -> pointer;
  }

  for (int i = 1; i <= n; i++) {
    printf("(x_{%i,1} OR ", i);
    for (int j = 2; j < n; j++) {
      printf("x_{%i,%i} OR ", i, j);
    }
    printf("x_{%i,%i}) AND\n", i, n);
  }

  for (int j = 1; j <= n; j++) {
    printf("(x_{1,%i} OR ", 1, j);
    for (int i = 2; i < n; i++) {
      printf("x_{%i,%i} OR ", i, j);
    }
    printf("x_{%i,%i} AND\n", n, j);
  }

  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      for (int k = 1; k <= n; k++) {
        if (j != k) {
          printf("NOT (x_{%i,%i} AND x_{%i,%i}) AND\n",
            i, j, i, k);
        }
      }
    }
  }

  for (int j = 1; j <= n; j++) {
    for (int i = 1; i <= n; i++) {
      for (int k = 1; k <= n; k++) {
        if (i != k) {
          printf("NOT (x_{%i,%i} AND x_{%i,%i}) AND\n",
            i, j, k, j);
        }
      }
    }
  }

  for (u = 1; u <= n; u++) {
    for (v = 1; v <= n; v++) {
      for (int j = 1; j < n; j++) {
        if (M[u][v] == 0) {
          printf("NOT (x_{%i,%i} AND x_{%i,%i}) AND\n",
            u, j, v, j + 1);
        }
      }
    }
  }

  printf("true\n");

  return(0);
}
```

Assignment 7 Solutions

1. You may have seen the standard BFS-based algorithm that, given a set of n jugs with capacities $c_1, \ldots, c_n$ in litres, a target of t litres and access to a tap, finds the fastest way of using the jugs to measure out t litres of water, or that it's not possible. If the c_is are integers then this takes $O((c_1 + 1) \cdot (c_2 + 1) \cdots (c_n + 1) \cdot (n + 1)^2)$ time.

   How can you modify the standard algorithm such that it runs in

   $$O((c_1 + 1) \cdot (c_2 + 1) \cdots (c_n + 1) \cdot (n + 1)^2 \cdot \log((c_1 + 1) \cdot (c_2 + 1) \cdots (c_n + 1))$$

   time and, instead of the fastest way (fewest pours), it finds the way to measure out t litres which involves the least total lifting? For example, pouring 2 litres from 5-litre jug containing 4 litres into a 3-litre jug containing 1 litre, and then pouring the remaining 2 litres from the 5-litre jug into a 6-litre jug containing 1 litre, involves lifting a jug containing 4 litres and then lifting a jug (the same one) that contains 2 litres, so the total cost is 6. (You can assume the tap has a hose so you can fill a jug without lifting it.)

   Solution: We run Dijstra's algorithm on the graph whose vertices are the $(c_1 + 1) \cdot (c_2 + 1) \cdots (c_n + 1)$ states of the jugs (the ith just can contain between 0 and c_i litres) and whose edges are the possible operations. There is an edge from u to v with weight $w > 0$ if we can change the state of the jugs from u to v with a pour from a jug containing w litres; there is an edge from u to v with weight 0 if we can change the state from u to v by filling a jug. There are at most $(n + 1)^2$ outgoing edges from each vertex, since each operation can be expressed as a pair in $\{1, \ldots, n\} \cup NULL \times \{1, \ldots, n\} \cup NULL$, with (i, j) indicating we pour water from the ith jug into the jth until the ith is empty or the jth is full, $(i, NULL)$ indicating we pour out the contents of the ith jug, and $(NULL, i)$ indicating we fill the ith jug. Therefore, Dijkstra's algorithm takes time

   $$O((c_1 + 1) \cdot (c_2 + 1) \cdots (c_n + 1) \cdot (n + 1)^2 \cdot \log((c_1 + 1) \cdot (c_2 + 1) \cdots (c_n + 1)) .$$

2. Suppose you're given a list of statements such as "FACTORING polytime reduces to SAT", "CLIQUE polytime reduces to INDEPENDENT SET", "INDEPENDENT SET polytime reduces to SAT" and "SAT polytime reduces to CLIQUE". How can you divide the mentioned problems up into the minimum number of equivalence classes such that, for any equivalence class C and any two problems P and Q in C, you know only from the statements that P polytime reduces to Q and vice versa? (In the example above, there are two equivalence classes: $\{$FACTORING$\}$ and $\{$SAT, CLIQUE, INDEPENDENT SET$\}$.)

   Solution: We build a graph whose vertices are mentioned problems, with an edge from u to v if one of the statements says u polytime reduces to v, and then find the strongly-connected components of that graph. Each strongly-connected component is an equivalence class.

3. How can you modify Dijkstra's (without making it much slower) such that it works even when there's *one* directed negative-weight edge in the graph?

   Solution: Suppose the graph is G and the negative-weight edge is from u to v and has cost $-w$. We use Dijkstra's algorithm to find the the shortest paths in $G - (u, v)$ from the starting vertex s to u, and from v to u.

- If there is no way to get from s to u, then we just run Dijkstra's on G.
- If there is a way to get from s to u and total weight of the shortest path from v to u is less than w, then there is a negative-weight cycle reachable from s, so the distance from s to any vertex reachable from v is undefined, and the distance from s to any other vertex is the same as in $G - (u, v)$.
- If there is a way to get from s to u and the total weight of the shortest path from v to u is w or more, then we replace (u, v) by an edge (s, v) whose weight is the distance from s to u in $G - (u, v)$ minus w, and run Dijkstra's on $G - (u, v) + (s, v)$.

4. How can you determine if there's a negative-weight cycle of length at most k in time $O(kn^3)$, where n is the number of vertices in the graph?

   Solution: We use repeated matrix multiplication (with min and + replacing + and $\times$, respectively) to compute the minimum distance from the ith to jth vertices across exactly ℓ edges, for $1 \leq i, j \leq n$ and $1 \leq \ell \leq k$. The distance from the ith vertex to itself across exactly ℓ edges is negative, for some $i \leq n$ and $\ell \leq k$, if and only if there is a negative-weight cycle of length at most k.

5. Give an $O(n^3 \log k)$-time algorithm that, given an integer k and the $n \times n$ adjacency matrix of a graph on n vertices, returns the $n \times n$ matrix in which cell (i, j) is the number of ways of going from the ith vertex to the jth vertex in exactly k steps.

   (Hint: First figure out how to do this when k is a power of 2, and then consider the binary representation of general k.)

   Solution: We start with the binary adjacency matrix M of the graph. Assume each cell (h, i) of matrix M^a stores the number of ways to go from the hth to the ith vertices in exactly a steps, and each cell (i, j) of matrix M^b stores the number of ways to go from the ith to the jth vertices in exactly b steps. (This is clearly true for $a = b = 1$, for simple graphs.) The number of ways to go from the hth vertex to the jth vertex in exactly $a + b$ steps is the sum over i of the number of ways to go from the hth vertex to the ith vertex in exactly a steps, times the number of ways to go from the ith vertex to the jth vertex in exactly b steps. This sum is the number in the cell (h, j) of $M^a M^b$. (So, by induction, our assumption is true for simple graphs.) Therefore, in $O(kn^3)$ time we can compute M^k, by k repeated matrix multiplications. To speed this up, we can use repeated squaring to compute M^{2^i} for $0 \leq i \leq \lfloor \lg k \rfloor$. To obtain M^k, we multiply together all the matrices M^{2^i} such that the ith bit from the right (counting from 0) in the binary representation of k is a 1. (I guess we could also speed up the computation with Strassen's algorithm.)

Assignment 8 Solutions

1. Consider the map from Assignment 1, shown below. Suppose you and lots of your friends have decided to celebrate the end (?) of the pandemic with trips from Dal to Eastern Europe. To maintain the feeling that the world is big and wide, you don't want to travel in big groups, nor keep running into each other while you're travelling.

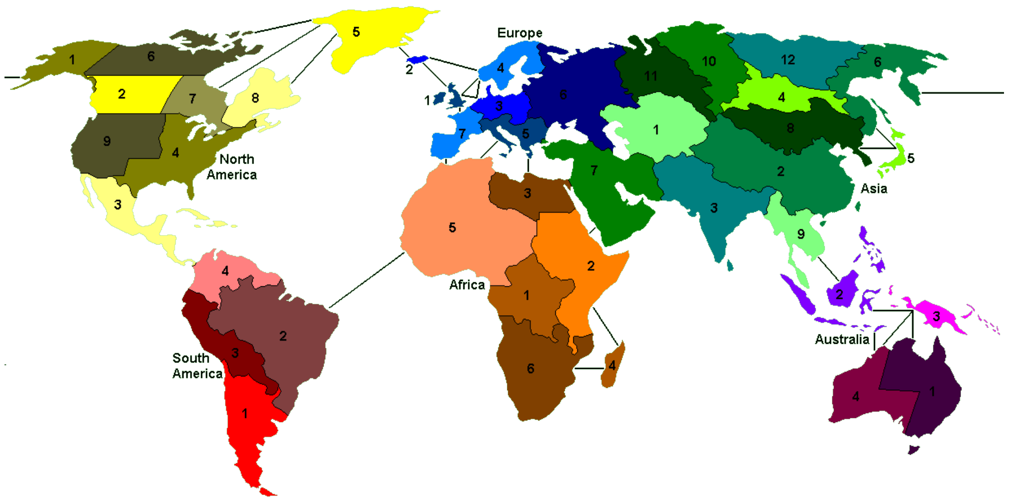
   You've decided to stay spread out with the following rule: on any particular day, if region A is labelled with the number x and region B is labelled with the number y, then at most $\min(x, y)$ people can travel from A to B, and at most $\min(x, y)$ can travel from B to A. (You can assume movements are synchronous.)

   The first few days may be a little chaotic, but then things should stabilize for a while, with the same number of people leaving each region as entering, until the last people are leaving Dal and eventually making their way to Eastern Europe. During this intermediate period of stability, how many people are leaving Dal each day (or, equivalently, how many are arriving in Eastern Europe)?

   (Hint: can you find a matching cut?)

   Each day,
   - 2 people go from Atlantic Canada to Greenland,
   - 2 people go from Greenland to Iceland,
   - 2 people go from Iceland to Norway / Sweden,
   - 2 people go from Norway / Sweden to Eastern Europe,
   - 1 person goes from Atlantic Canada to Ontario,
   - 1 person goes from Ontario to Western Canada,
   - 1 person goes from Western Canada to Alaska,
   - 1 person goes from Alaska to Kamchatka,
   - 1 person goes from Kamchatka to Yakutsk,
   - 1 person goes from Yakutsk to Siberia,
   - 1 person goes from Siberia to the Urals,
   - 1 person goes from the Urals to Eastern Europe,
   - 2 people go from Atlantic Canada to the Eastern US,
   - 2 people go from the Eastern US to Central America,
   - 2 people go from Central America to northern South America,
   - 2 people go from northern South America to Brazil,
   - 2 people go from Brazil to West Africa,
   - 2 people go West Africa to Southern Europe,
   - 2 people go from Southern Europe to Eastern Europe,

   so 5 people leave Atlantic Canada and 5 people arrive in Eastern Europe. This is the most possible, because the lines from Greenland to Iceland, from Alaska to Kamchatka and from Brazil to West Africa are a cut between Atlantic Canada and Eastern Europe with total capacity 5.



2. Suppose you're organizing a dinner at an event with n students from k universities and trying to choose a seating plan. Eight people can sit at each table and you don't want more than three people from the same university sitting at the same table. How can you efficiently find the minimum number of tables you'll need? The input is the number of people from each university, $n_1, \ldots, n_k$ with $n_1 + \cdots + n_k = n$, and the output is the number of tables.

   (Hint: to test if t tables are enough, create a graph G_t with a source, a sink, a vertex for each university, and a vertex for each of t tables.)

   Solution: We start with $t = 1$ and incrementing t until we find a solution with everyone seated. For each value t, we build a graph with a source vertex s, k vertices $u_1, \ldots, u_k$, t vertices $v_1, \ldots, v_t$, and a sink vertex t. We add an edge between s and each u_i with capacity n_i; an edge between each vertex u_i and each vertex v_j with capacity 3; and an edge between each vertex v_j and t with capacity 8. We then find a maximum flow in the graph with Ford-Fulkerson algorithm, so the flow on each edge is integer. If the flow is n, then everyone is seated: we put as many people from the ith university at the jth table as there is flow on the edge (u_i, v_j).

3. We saw in the Lecture 19 that if we assume there's a C routine P that, given any string S, returns the length in characters of the shortest C program that outputs S and then stops, then we reach a contradiction.

   Specifically, if the code for P looks like

```
int P (char *S) {
  ...SOME CODE GOES HERE...
}
```

   (with the code to compute P replacing `...SOME CODE GOES HERE...`), then we can write a program Q that has P as a subroutine and loops through all possible strings in increasing order by length until it finds one that P says requires a program much longer than Q, at which point Q stops and outputs that string. (P must eventually say some string requires a program much longer than Q, by counting arguments.) An example of such a program Q is shown below.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct node {
  char *string;
  struct node *next;
} node;

char *stringP = "int P (char *S) {\n  ...SOME CODE GOES HERE...\n}";
int lenP = strlen(stringP);

int P (char *S) {
  ...SOME CODE GOES HERE...
}

int main() {
  node *head = (node *) malloc(sizeof(node));
  node *tail = head;

  tail -> string = (char *) malloc(1);
  sprintf(tail -> string, "");
  tail -> next = NULL;

  while (1) {
    if (P(head -> string) > 2 * lenP + 1000000) {
    // Notice P appears twice in this program:
    // once as a string and once as a subroutine.
    // The number 1000000 only has to be more than
    // the length of the rest of the program.
      printf("%s", head -> string);
      return(0);
    } else {
      for (int c = 0; c < 256; c++) {
        tail -> next = (node *) malloc(sizeof(node));
        tail = tail -> next;
        tail -> string = (char *) malloc(strlen(head -> string) + 2);
        sprintf(tail -> string, "%s%c", head -> string, (char) c);
        tail -> next = NULL;
      }
      head = head -> next;
    }
  }
}
```

Adapt that argument to show it's not possible even to *approximate within a factor of 10* the length of the shortest C program that outputs S and then stops. How does the code for Q above change?

Solution: Assume P approximates within a factor of 10 the length of the shortest C program that outputs S and then stops. If we change the line

```
if (P(head -> string) > 2 * lenP + 1000000) {
```

to

```
if (P(head -> string) > 10 * (2 * lenP + 1000000)) {
```

in the program above, then the program loops through all possible strings in increasing order by length until it finds one that P says requires a program more than 10 times longer than the program itself, at which point it outputs that string and stops. Since this contradicts our assumption, P doesn't exist.

Final Exam Solutions

1. Suppose you are given a rooted tree T on n vertices, each of which is assigned a colour. (In this problem, both endpoints of an edge can have the same colour.) Give an efficient divide-and-conquer algorithm to find the longest path in T whose vertices are all the same colour. (Paths can ascend and then descend, as long as they don't revisit vertices.) What is the complexity of your algorithm? **You need not prove your algorithm correct.**

   Solution: Working bottom-up, for each vertex v of T we compute
   - the longest monochromatic path in v's subtree that starts at v,
   - the longest monochromatic path in v's subtree that includes v,
   - the longest monochromatic path in v's subtree.

   If v is a leaf, then all three of these paths are just v itself.
   If v is an internal vertex with no children the same colour as v, then the longest monochromatic path in v's subtree that starts at v, is just v itself.
   If v is an internal vertex with at least one child the same colour as v, we find the longest monochromatic path in v's subtree that starts at v by finding the child w of v such that
   - w is the same colour as v,
   - the longest monochromatic path in w's subtree that starts at w is at least as long as the longest monochromatic path in x's subtree that starts at x, for any other child x of v that is the same colour as v.

   We prepend (v, w) to the longest monochromatic path in w's subtree that starts at w.
   If v has at most one child the same colour as itself, then the longest monochromatic path in v's subtree that includes v is the longest monochromatic path in v's subtree that starts at v. Otherwise, let w and x be the two children of v with the same colour as v, that have the two longest monochromatic paths P_w and P_x in their subtrees and starting at them. The longest monochromatic path in v's subtree that includes v is the reverse of P_w, followed by (w, v), followed by (v, x), followed by P_x.
   To find the longest monochromatic path in v's subtree, we compare the longest monochromatic path in v's subtree that includes v, to the longest monochromatic path in w's subtree, for each child w of v; we choose the longest of all those paths.
   This algorithm takes $O(n)$ time since, for each vertex v, the time we spend at v is proportional to the number of v's children.
   (Students' solutions needn't be as detailed as this; as long as it's clear they understand the basic idea, they should get most of the marks. Some students may also say they delete all the edges whose endpoints have different colours, and then find the longest path in the resulting forest, ignoring colours. As long as they correctly say how to find the longest path in a tree — using divide-and-conquer! — that's fine too. If that explanation is missing, some marks should be deducted.)

2. (a) Give an efficient greedy algorithm that, given a sequence $A[1..n]$ of integers, partitions A into the minimum number of kind-of increasing subsequences. A subsequence is *kind-of increasing* if, for each consecutive pair of numbers $A[h]$ and $A[i]$ in the subsequence, with $h < i$, we have $A[h] - 3 \leq A[i]$. **Prove your algorithm correct.**

Solution: As with Supowit's algorithm, we build the subsequences as linked lists, with the last integer of each list in a dynamic predecessor data structure, such as an AVL tree. We process A from left to right and, for each number $A[i]$, we use the predecessor structure to find the largest integer x currently at the end of a list such that $x - 3 \leq A[i]$; append $A[i]$ to x's list; delete x from the predecessor data structure; and insert $A[i]$. If there is no such number x, we create a new list initially containing only $A[i]$, and insert $A[i]$ into the predecessor data structure.

Before we take any steps, our empty subsolution can be extended to an optimal solution. Suppose that after i steps our subsolution can be extended to an optimal solution S. We show that after $i + 1$ steps our subsolution can be extended to an optimal solution S'.

If none of the current lists end with a number x such that $x - 3 \leq A[i]$ when we process $A[i + 1]$ in the $(i + 1)$st step, then S too must start a new subsequence with $A[i + 1]$, so S extends our subsolution after $i + 1$ steps.

If S also puts $A[i + 1]$ after the largest integer x currently at the end of a list such that $x - 3 \leq A[i + 1]$, then S extends our subsolution after $i + 1$ steps.

If S puts $A[i + 1]$ after another integer y, then $y - 3 \leq x - 3 \leq A[i + 1]$. Let α be the rest of the subsequence after y in S, and let β be the rest of the subsequence after x in S. Since α starts with $A[i + 1]$, we can move α to after x; since the first integer in β is at least $x - 3 \geq y - 3$, we can move β to after y. This results in a solution S' which extends our subsolution after $i + 1$ steps, and which has no more subsequences than S and is thus also optimal.

(b) What goes wrong if you try to partition S into the number of slowly increasing subsequences the same way? A subsequence is *slowly increasing* if, for each consecutive pair of numbers $S[i]$ and $S[j]$ in the subsequence, with $i < j$, $S[i] < S[j] \leq S[i] + 10$?

(Hint: consider the two sequences 8, 1, 9, 2 and 8, 1, 9, 18, 18.)

Solution: The exchange argument in the proof breaks down. In the example from the hint, after we process the 1 we must have two partial subsequences, 8 and 1. If we put the 9 after 1 and then we see a 2, we must start a new subsequence with that 2, even though the optimal partition is 8-9 and 1-2. If we put the 9 after the 8 and then we see 18 and 18, then we can put one copy of 18 after the 9 but must start another subsequence with the other 18, even though the optimal partition is 8-18 and 1-9-18.

3. Suppose you are given a directed acyclic graph G on n vertices, each of which is assigned a colour. (In this problem, both endpoints of an edge can have the same colour.) Give an efficient dynamic-programming algorithm to find the length of the longest directed path in G whose vertices are all the same colour. What is the complexity of your algorithm? **You need not prove your algorithm correct.**

(Hint: start with a topological sort.)

Solution: We first perform a topological sort of G and number the vertices from 1 to n such that, if there is an edge from v to w, then v's number is *bigger* than w's number. We build an array $A[1..n]$ such that $A[i]$ is the length of the longest monochromatic directed path starting at the vertex numbered i.

If the ith vertex v is a leaf or there are no edges from v to other vertices the same colour as v, then the longest monochromatic directed path starting at v is just v itself, so $A[i] = 0$. Otherwise, the length of the longest monochromatic directed path starting at v is

$$\max\{A[h]+1 \ : \text{ the } h\text{th vertex } w \text{ is the same colour as } v \text{ and there is an edge from } v \text{ to } w\}\,.$$

This algorithm takes $O(m+n)$ time, where m is the number of edges and n is the number of vertices.
(Again, some students may say they delete edges whose endpoints are different colours, and then find the longest directed path in the resulting DAG, ignoring colours. That's fine as long as they correctly say how to find the longest directed path in a DAG using dynamic programming.)

4. Suppose you are given a graph G on n vertices, each of which is assigned a colour. (In this problem, both endpoints of an edge can have the same colour.) Either give an efficient algorithm to find the longest monochromatic path in G or justify your inability to do so. **If you give an algorithm, you need not prove it correct.**

   Solution: This is NP-hard, by a reduction from HAMPATH. Suppose we are given a graph G and asked if there is a Hamiltonian path. If we colour all the vertices of G the same colour, then there is a monochromatic path of length $n-1$ if and only if there is a Hamiltonian path.

5. We reduced 3-COL to PLANAR 3-COL in polynomial time using a *crossing gadget*. Why can't we reduce 4-COL to PLANAR 4-COL in polynomial time the same way (assuming P $\neq$ NP)? What goes wrong in the reduction?

   Solution: Assuming P $\neq$ NP, there is no polynomial time reduction from 4-COL (which is NP-complete) to PLANAR 4-COL (which is in P, by the 4-Colour Theorem). We can't use the same crossing gadget from the reduction from 3-COL to PLANAR 3-COL because the leftmost and rightmost vertices are the same colour, and the topmost and bottommost vertices are the same colour, when the gadget is 3-coloured, not when it's 4-coloured.

6. Suppose you are given a *directed* graph G on n vertices, each of which is assigned a colour. (In this problem, both endpoints of an edge can have the same colour.) Give an efficient algorithm to find the length of the longest monochromatic directed walk in G. (In a walk we can revisit vertices and recross edges, whereas in a path we cannot.) Your algorithm should return "undefined" if there is a directed cycle whose vertices are all the same colour.

   Solution: For each colour, we consider the induced subgraph on the vertices with that colour (that is, the subgraph remaining after deleting all edges except those with both endpoints that colour, and deleting all vertices of different colours). We check each such subgraph to see if it contains a directed cycle (using the topological-sort algorithm, for example) and, if so, we return "undefined". Otherwise, we use our solution to Question 3 on that subgraph, and return the length of the longest directed path in any of the subgraphs.
   (This can also be done with matrix multiplication.)